* **So we are learning about ble communication and we use arduino nano 33 ble and laptop for that. I have arduino and laptop code that both work. Now i want to use 2nd arduino nano in communication. The plan is to send json rpc from laptop in which we will write from which arduino we want to retrieve sensors readings (e.g. Device1, device2). Could you rewrite the given code?**

Arduino code:

#include <ArduinoBLE.h>

#include <ArduinoJson.h>

// Simulated sensor readings for demo purposes

float readTemperature() { return 22.5; } // Replace with actual sensor read function

float readHumidity()    { return 60.2; } // Replace with actual sensor read function

BLEService jsonRpcService("91ED0001-0000-0000-0000-000000000000"); // Custom UUID for JSON-RPC service

BLECharacteristic jsonRpcCharacteristic("91ED0002-0000-0000-0000-000000000000", BLERead | BLEWrite, "json-rpc");

void setup() {

  Serial.begin(9600);

  while (!Serial);

  // Initialize BLE

  if (!BLE.begin()) {

    Serial.println("Starting BLE failed!");

    while (1);

  }

  // Set advertised local name and service

  BLE.setLocalName("Nano33BLE-JSON-RPC");

  BLE.setAdvertisedService(jsonRpcService);

  // Add the characteristic to the service

  jsonRpcService.addCharacteristic(jsonRpcCharacteristic);

  // Add service to BLE

  BLE.addService(jsonRpcService);

  // Initialize characteristic with empty value

  jsonRpcCharacteristic.writeValue("");

  // Start advertising

  BLE.advertise();

  Serial.println("BLE JSON-RPC Service Started. Waiting for connections...");

}

void loop() {

  BLE.poll();

  // Check if a device is connected

  if (BLE.connected()) {

    // Check if the characteristic's value has been written to (i.e., incoming JSON-RPC request)

if (jsonRpcCharacteristic.written()) {

      size\_t jsonDataLength = jsonRpcCharacteristic.valueLength();

      const char\* jsonData = (const char\*)jsonRpcCharacteristic.value(); // Cast to const char\*

      String incomingJson = String(jsonData, jsonDataLength); // Safe conversion to String

      Serial.println("Incoming JSON-RPC Request:");

      Serial.println(incomingJson);

      DynamicJsonDocument incomingDoc(2048);

      DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

      if (error) {

        Serial.print("deserializeJson() failed: ");

        Serial.println(error.f\_str());

        respondWithError("Parse error", 1); // Assuming id is always present for simplicity

      } else {

        if (incomingDoc["method"] == "getSensorReadings") {

          respondWithSensorReadings(incomingDoc["id"]);

        } else {

          respondWithError("Method not found", incomingDoc["id"]);

        }

      }

    }

  }

}

void respondWithError(const char\* errorMessage, int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["error"]["code"] = -32600; // Method not found

  outgoingDoc["error"]["message"] = errorMessage;

  outgoingDoc["id"] = id;

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length()); // Convert String to const char\* and specify length

  Serial.println("Sent Error Response:");

  Serial.println(outgoingJson);

}

void respondWithSensorReadings(int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["result"]["temperature"] = readTemperature();

  outgoingDoc["result"]["humidity"] = readHumidity();

  outgoingDoc["id"] = id;

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length()); // Convert String to const char\* and specify length

  Serial.println("Sent Sensor Readings Response:");

  Serial.println(outgoingJson);

}

Laptop code:

import asyncio

from bleak import BleakClient

import json

Service and Characteristic UUIDs (Match those in the Arduino code)

JSON\_RPC\_SERVICE\_UUID = "91ED0001-0000-0000-0000-000000000000"

JSON\_RPC\_CHARACTERISTIC\_UUID = "91ED0002-0000-0000-0000-000000000000"

Arduino Nano 33 BLE Device Name

TARGET\_DEVICE\_NAME = "Nano33BLE-JSON-RPC"

async def main():

    # Scan for devices

    print("Scanning for devices...")

    devices = await BleakClient.discover()

    target\_device = next((d for d in devices if d.name == TARGET\_DEVICE\_NAME), None)

    if not target\_device:

        print(f"Failed to find {TARGET\_DEVICE\_NAME} in nearby devices.")

        return

    print(f"Found {TARGET\_DEVICE\_NAME} - {target\_device.address}")

    # Connect to the device

    async with BleakClient(target\_device) as client:

        print(f"Connected to {TARGET\_DEVICE\_NAME}")

        # Discover the service and characteristic

        svc = await client.get\_service(JSON\_RPC\_SERVICE\_UUID)

        if not svc:

            print(f"Failed to find service {JSON\_RPC\_SERVICE\_UUID}")

            return

        char = next((c for c in svc.characteristics if c.uuid == JSON\_RPC\_CHARACTERISTIC\_UUID), None)

        if not char:

            print(f"Failed to find characteristic {JSON\_RPC\_CHARACTERISTIC\_UUID} in service")

            return

        # Prepare and send the JSON-RPC request

        request\_id = 1

        json\_rpc\_request = {

            "jsonrpc": "2.0",

            "method": "getSensorReadings",

            "params": [],

            "id": request\_id

        }

        request\_json = json.dumps(json\_rpc\_request)

        print(f"Sending JSON-RPC Request: {request\_json}")

        await client.write\_gatt\_char(char, request\_json.encode())

        # Wait for the response (Note: This example assumes the response comes immediately after the request.

        #  In a real-world scenario, consider implementing a more robust waiting mechanism.)

        await asyncio.sleep(1)  # Wait for 1 second

        # Read the response (Assuming it's available in the characteristic value)

        response = await client.read\_gatt\_char(char)

        response\_json = response.decode()

        print(f"Received JSON-RPC Response: {response\_json}")

        try:

            response\_data = json.loads(response\_json)

            if "result" in response\_data:

                print(f"Temperature: {response\_data['result']['temperature']}°C, Humidity: {response\_data['result']['humidity']}%")

            else:

                print("No sensor readings in response.")

        except json.JSONDecodeError as e:

            print(f"Failed to parse JSON response: {e}")

asyncio.run(main())

* **Something is wrong. Could you first write an arduino code?**

Here is a rewritten Arduino code for **both** Arduino Nano 33 BLE devices, with the requested functionality to handle JSON-RPC requests that specify the target device by name:

**Arduino Code (identical for both devices, except for the LOCAL\_NAME definition)**

#include <ArduinoBLE.h>

#include <ArduinoJson.h>

// \*\*CHANGE THIS FOR EACH DEVICE\*\*

const char\* LOCAL\_NAME = "Nano33BLE-JSON-RPC-1"; // Use "Nano33BLE-JSON-RPC-2" for the second device

// Simulated sensor readings for demo purposes

float readTemperature() { return 22.5; } // Replace with actual sensor read function

float readHumidity() { return 60.2; } // Replace with actual sensor read function

BLEService jsonRpcService("91ED0001-0000-0000-0000-000000000000");

BLECharacteristic jsonRpcCharacteristic("91ED0002-0000-0000-0000-000000000000", BLERead | BLEWrite, "json-rpc");

void setup() {

Serial.begin(9600);

while (!Serial);

if (!BLE.begin()) {

Serial.println("Starting BLE failed!");

while (1);

}

BLE.setLocalName(LOCAL\_NAME);

BLE.setAdvertisedService(jsonRpcService);

jsonRpcService.addCharacteristic(jsonRpcCharacteristic);

BLE.addService(jsonRpcService);

jsonRpcCharacteristic.writeValue("");

BLE.advertise();

Serial.println("BLE JSON-RPC Service Started. Waiting for connections...");

}

void loop() {

BLE.poll();

if (BLE.connected()) {

if (jsonRpcCharacteristic.written()) {

size\_t jsonDataLength = jsonRpcCharacteristic.valueLength();

const char\* jsonData = (const char\*)jsonRpcCharacteristic.value();

String incomingJson = String(jsonData, jsonDataLength);

Serial.println("Incoming JSON-RPC Request:");

Serial.println(incomingJson);

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (error) {

Serial.print("deserializeJson() failed: ");

Serial.println(error.f\_str());

respondWithError("Parse error", 1); // Assuming id is always present for simplicity

} else {

if (incomingDoc["method"] == "getSensorReadings") {

const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device

if (strcmp(targetDevice, LOCAL\_NAME) == 0) { // If request is for this device

respondWithSensorReadings(incomingDoc["id"]);

} else { // Ignore requests not meant for this device

Serial.println("Ignoring request not meant for this device.");

}

} else {

respondWithError("Method not found", incomingDoc["id"]);

}

}

}

}

}

void respondWithError(const char\* errorMessage, int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["error"]["code"] = -32600; // Method not found

outgoingDoc["error"]["message"] = errorMessage;

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Error Response:");

Serial.println(outgoingJson);

}

void respondWithSensorReadings(int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["result"]["temperature"] = readTemperature();

outgoingDoc["result"]["humidity"] = readHumidity();

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Sensor Readings Response:");

Serial.println(outgoingJson);

}

**To use:**

1. Upload this code to **both** Arduino Nano 33 BLE devices.
2. For the **second device**, change the LOCAL\_NAME definition at the top to "Nano33BLE-JSON-RPC-2".
3. Ensure both devices are powered and in range.
4. Proceed with modifying the Laptop code (which I'll provide next if this Arduino code works as expected for you). The Laptop code will need to be adjusted to specify the target device in the JSON-RPC request.

* **Do I need to change BLEservice or characteristic address for second arduino?**

**Short Answer:** No, you don't necessarily need to change the BLE Service or Characteristic UUIDs for the second Arduino, but it's **highly recommended** to do so for a production environment or if you plan to have more than two devices in the same vicinity. Here's why:

**Reasons to keep the same UUIDs for development/testing:**

1. **Simplified Development**: Using the same UUIDs for both devices simplifies your development process, as you won't need to modify the Laptop code to accommodate different service and characteristic UUIDs for each device.
2. **Easier Testing**: Testing becomes more straightforward since the Laptop code remains unchanged, and you can focus on verifying the device differentiation logic based on the LOCAL\_NAME or the targetDevice parameter in the JSON-RPC request.

**Reasons to change UUIDs for production or multiple devices:**

1. **Uniqueness**: UUIDs are designed to be universally unique. Using the same UUIDs for multiple devices, especially in a production environment, goes against this principle and might lead to unforeseen conflicts or issues, particularly if devices are in close proximity.
2. **Device Differentiation**: While using LOCAL\_NAME for differentiation works for simple scenarios, relying solely on this for multiple devices in production might not be robust enough. Unique UUIDs for services and characteristics provide an additional, more inherent layer of device differentiation.
3. **Future Scalability**: If you anticipate adding more devices or features, using unique UUIDs from the outset makes your system more scalable and easier to manage.

**How to generate unique UUIDs:**

* **Online UUID Generators**: Utilize online tools to generate truly unique UUIDs.
* **BLE UUID Registration**: For production environments, consider registering your UUIDs with the Bluetooth Special Interest Group (SIG) to ensure global uniqueness.

**Modified Code for Unique UUIDs (Example):**

If you decide to use unique UUIDs for each device, you would change the following lines in the Arduino code for the **second device**:

BLEService jsonRpcService("91ED0003-0000-0000-0000-000000000000"); // New UUID for the second device

BLECharacteristic jsonRpcCharacteristic("91ED0004-0000-0000-0000-000000000000", BLERead | BLEWrite, "json-rpc"); // New UUID for the characteristic

**Remember**, if you change the UUIDs, you'll also need to update the Laptop code to match the new UUIDs for device discovery and communication.

* **Could you now give me a laptop code which should work with arduino code that you provided earlier?**

Here's a Python script using the bleak library that should work with the Arduino code I provided earlier. This script allows you to specify the target device by its LOCAL\_NAME (e.g., "Nano33BLE-JSON-RPC-1" or "Nano33BLE-JSON-RPC-2") and send a JSON-RPC request to retrieve sensor readings.

**Laptop Code (Python):**

import asyncio

from bleak import BleakClient

import json

# Service and Characteristic UUIDs (Match those in the Arduino code)

JSON\_RPC\_SERVICE\_UUID = "91ED0001-0000-0000-0000-000000000000"

JSON\_RPC\_CHARACTERISTIC\_UUID = "91ED0002-0000-0000-0000-000000000000"

# Arduino Nano 33 BLE Device Names

TARGET\_DEVICE\_NAMES = ["Nano33BLE-JSON-RPC-1", "Nano33BLE-JSON-RPC-2"]

async def main():

# Scan for devices

print("Scanning for devices...")

devices = await BleakClient.discover()

# Find target devices

target\_devices = {device.name: device for device in devices if device.name in TARGET\_DEVICE\_NAMES}

if not target\_devices:

print("No target devices found.")

return

# Select target device

print("Select target device:")

for i, (name, device) in enumerate(target\_devices.items()):

print(f"{i+1}. {name} - {device.address}")

choice = input("Enter the number of your choice: ")

try:

choice = int(choice)

if choice < 1 or choice > len(target\_devices):

print("Invalid choice.")

return

except ValueError:

print("Invalid input. Please enter a number.")

return

selected\_name = list(target\_devices.keys())[choice - 1]

target\_device = target\_devices[selected\_name]

print(f"Connecting to {selected\_name} - {target\_device.address}...")

async with BleakClient(target\_device) as client:

print(f"Connected to {selected\_name}")

# Discover the service and characteristic

svc = await client.get\_service(JSON\_RPC\_SERVICE\_UUID)

if not svc:

print(f"Failed to find service {JSON\_RPC\_SERVICE\_UUID}")

return

char = next((c for c in svc.characteristics if c.uuid == JSON\_RPC\_CHARACTERISTIC\_UUID), None)

if not char:

print(f"Failed to find characteristic {JSON\_RPC\_CHARACTERISTIC\_UUID} in service")

return

# Prepare and send the JSON-RPC request

request\_id = 1

target\_device\_name = selected\_name # Use the selected device's name in the request

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": "getSensorReadings",

"params": [{"device": target\_device\_name}], # Specify the target device

"id": request\_id

}

request\_json = json.dumps(json\_rpc\_request)

print(f"Sending JSON-RPC Request: {request\_json}")

await client.write\_gatt\_char(char, request\_json.encode())

# Wait for the response (Note: This example assumes the response comes immediately after the request.

# In a real-world scenario, consider implementing a more robust waiting mechanism.)

await asyncio.sleep(1) # Wait for 1 second

# Read the response (Assuming it's available in the characteristic value)

response = await client.read\_gatt\_char(char)

response\_json = response.decode()

print(f"Received JSON-RPC Response: {response\_json}")

try:

response\_data = json.loads(response\_json)

if "result" in response\_data:

print(f"Temperature: {response\_data['result']['temperature']}°C, Humidity: {response\_data['result']['humidity']}%")

else:

print("No sensor readings in response.")

except json.JSONDecodeError as e:

print(f"Failed to parse JSON response: {e}")

asyncio.run(main())

**How to use:**

1. Ensure you have Python and the bleak library installed (pip install bleak).
2. Run the script. It will scan for nearby BLE devices and list the target devices found.
3. Select the device you wish to connect to by entering its corresponding number.
4. The script will send a JSON-RPC request to the selected device and print the response.

**Traceback (most recent call last):**

**File "/Users/anteseric/Desktop/llama/two\_devices\_laptop\_code.py", line 91, in <module>**

**asyncio.run(main())**

**~~~~~~~~~~~^^^^^^^^**

**File "/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/asyncio/runners.py", line 194, in run**

**return runner.run(main)**

**~~~~~~~~~~^^^^^^**

**File "/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/asyncio/runners.py", line 118, in run**

**return self.\_loop.run\_until\_complete(task)**

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^**

**File "/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/asyncio/base\_events.py", line 721, in run\_until\_complete**

**return future.result()**

**~~~~~~~~~~~~~^^**

**File "/Users/anteseric/Desktop/llama/two\_devices\_laptop\_code.py", line 15, in main**

**devices = await BleakClient.discover()**

**^^^^^^^^^^^^^^^^^^^^**

**AttributeError: type object 'BleakClient' has no attribute 'discover'**

The error message indicates that the BleakClient class has no attribute discover. This is because the discover method has been moved to a separate class called BleakScanner in newer versions of the bleak library.

Here's an updated version of your laptop code that should work with the latest bleak library:

import asyncio

from bleak import BleakClient, BleakScanner

import json

# Service and Characteristic UUIDs (Match those in the Arduino code)

JSON\_RPC\_SERVICE\_UUID = "91ED0001-0000-0000-0000-000000000000"

JSON\_RPC\_CHARACTERISTIC\_UUID = "91ED0002-0000-0000-0000-000000000000"

# Arduino Nano 33 BLE Device Names

TARGET\_DEVICE\_NAMES = ["Nano33BLE-JSON-RPC-1", "Nano33BLE-JSON-RPC-2"]

async def main():

# Scan for devices

print("Scanning for devices...")

scanner = BleakScanner()

await scanner.start()

await asyncio.sleep(5) # Scan for 5 seconds

await scanner.stop()

# Find target devices

target\_devices = {device.name: device for device in scanner.discovered\_devices if device.name in TARGET\_DEVICE\_NAMES}

if not target\_devices:

print("No target devices found.")

return

# Select target device

print("Select target device:")

for i, (name, device) in enumerate(target\_devices.items()):

print(f"{i+1}. {name} - {device.address}")

choice = input("Enter the number of your choice: ")

try:

choice = int(choice)

if choice < 1 or choice > len(target\_devices):

print("Invalid choice.")

return

except ValueError:

print("Invalid input. Please enter a number.")

return

selected\_name = list(target\_devices.keys())[choice - 1]

target\_device = target\_devices[selected\_name]

print(f"Connecting to {selected\_name} - {target\_device.address}...")

async with BleakClient(target\_device) as client:

print(f"Connected to {selected\_name}")

# Discover the service and characteristic

svc = await client.get\_service(JSON\_RPC\_SERVICE\_UUID)

if not svc:

print(f"Failed to find service {JSON\_RPC\_SERVICE\_UUID}")

return

char = next((c for c in svc.characteristics if c.uuid == JSON\_RPC\_CHARACTERISTIC\_UUID), None)

if not char:

print(f"Failed to find characteristic {JSON\_RPC\_CHARACTERISTIC\_UUID} in service")

return

# Prepare and send the JSON-RPC request

request\_id = 1

target\_device\_name = selected\_name # Use the selected device's name in the request

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": "getSensorReadings",

"params": [{"device": target\_device\_name}], # Specify the target device

"id": request\_id

}

request\_json = json.dumps(json\_rpc\_request)

print(f"Sending JSON-RPC Request: {request\_json}")

await client.write\_gatt\_char(char, request\_json.encode())

# Wait for the response (Note: This example assumes the response comes immediately after the request.

# In a real-world scenario, consider implementing a more robust waiting mechanism.)

await asyncio.sleep(1) # Wait for 1 second

# Read the response (Assuming it's available in the characteristic value)

response = await client.read\_gatt\_char(char)

response\_json = response.decode()

print(f"Received JSON-RPC Response: {response\_json}")

try:

response\_data = json.loads(response\_json)

if "result" in response\_data:

print(f"Temperature: {response\_data['result']['temperature']}°C, Humidity: {response\_data['result']['humidity']}%")

else:

print("No sensor readings in response.")

except json.JSONDecodeError as e:

print(f"Failed to parse JSON response: {e}")

asyncio.run(main())

**Changes Made:**

1. Imported BleakScanner from bleak.
2. Created a BleakScanner instance to scan for devices.
3. Started the scanner using await scanner.start(), waited for 5 seconds using await asyncio.sleep(5), and then stopped the scanner using await scanner.stop().
4. Retrieved the discovered devices using scanner.discovered\_devices.

* **Is it possible somehow for laptop to not know a characteristic address that it try to access, could peripheral device assume which characteristic laptop try to access based on method in received json rpc?**

Yes, it's possible to design a system where the laptop (central device) doesn't need to know the characteristic address beforehand, and the peripheral device (Arduino in your case) can infer which characteristic to use based on the method in the received JSON-RPC request. Here's a high-level approach to achieve this:

**Modified System Design:**

1. **Single Characteristic for JSON-RPC**:
   * Expose only one characteristic on the peripheral device (e.g., Arduino) for JSON-RPC communication.
   * This characteristic will handle all incoming JSON-RPC requests.
2. **Method-Based Routing on Peripheral Device**:
   * On the peripheral device, implement a routing mechanism that inspects the method field in the received JSON-RPC request.
   * Based on the method, the peripheral device determines which internal function or sensor reading to invoke.
3. **Laptop (Central Device) JSON-RPC Request**:
   * The laptop constructs JSON-RPC requests with the method corresponding to the desired action (e.g., getTemperature, getHumidity, etc.).
   * The laptop sends these requests to the single, known characteristic on the peripheral device.

**Example Implementation on Peripheral Device (Arduino):**

void loop() {

// ...

if (jsonRpcCharacteristic.written()) {

// ...

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (!error) {

const char\* method = incomingDoc["method"];

if (strcmp(method, "getTemperature") == 0) {

respondWithTemperature(incomingDoc["id"]);

} else if (strcmp(method, "getHumidity") == 0) {

respondWithHumidity(incomingDoc["id"]);

} else if (strcmp(method, "getSensorReadings") == 0) {

respondWithSensorReadings(incomingDoc["id"]);

} else {

respondWithError("Method not found", incomingDoc["id"]);

}

} else {

// Handle deserialization error

}

}

}

// Example response functions

void respondWithTemperature(int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["result"]["temperature"] = readTemperature();

outgoingDoc["id"] = id;

// Send response

}

void respondWithHumidity(int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["result"]["humidity"] = readHumidity();

outgoingDoc["id"] = id;

// Send response

}

void respondWithSensorReadings(int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["result"]["temperature"] = readTemperature();

outgoingDoc["result"]["humidity"] = readHumidity();

outgoingDoc["id"] = id;

// Send response

}

**Laptop (Central Device) Example JSON-RPC Requests:**

# Request temperature

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": "getTemperature",

"id": 1

}

# Request humidity

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": "getHumidity",

"id": 2

}

# Request all sensor readings

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": "getSensorReadings",

"id": 3

}

By adopting this approach, the laptop doesn't need to know the specific characteristic addresses for each sensor reading. Instead, it relies on the method in the JSON-RPC request to determine the action on the peripheral device.

* **So i just need to add this part in previous arduino code and it should works, right?**

Almost!

To integrate the method-based routing into your previous Arduino code, you'll need to make a few adjustments. Here's a step-by-step guide to help you merge the two:

**1. Replace the if (incomingDoc["method"] == "getSensorReadings") block:**

* In your original Arduino code, find the if (incomingDoc["method"] == "getSensorReadings") block within the loop() function.
* Replace this entire block with the new method-based routing code I provided earlier.

**2. Add the new response functions:**

* Copy the respondWithTemperature(), respondWithHumidity(), and respondWithSensorReadings() functions from my previous example.
* Paste these functions into your Arduino code, ensuring they are outside the loop() function (e.g., below it).

**3. Update the respondWithError() function (optional):**

* If you want to provide more informative error messages, consider updating the respondWithError() function to include the unknown method name.
* Example: respondWithError("Unknown method", incomingDoc["method"], incomingDoc["id"]);

**4. Define the readTemperature() and readHumidity() functions:**

* Make sure you have the readTemperature() and readHumidity()functions defined in your Arduino code.
* These functions should return the actual temperature and humidity values, respectively.

Here's the updated loop() function with the new method-based routing:

void loop() {

// ...

if (jsonRpcCharacteristic.written()) {

// ...

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (!error) {

const char\* method = incomingDoc["method"];

if (strcmp(method, "getTemperature") == 0) {

respondWithTemperature(incomingDoc["id"]);

} else if (strcmp(method, "getHumidity") == 0) {

respondWithHumidity(incomingDoc["id"]);

} else if (strcmp(method, "getSensorReadings") == 0) {

respondWithSensorReadings(incomingDoc["id"]);

} else {

respondWithError("Unknown method", incomingDoc["method"], incomingDoc["id"]);

}

} else {

// Handle deserialization error

}

}

}

**Updated respondWithError() function (optional):**

void respondWithError(const char\* errorMessage, const char\* methodName, int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["error"]["code"] = -32601; // Method not found

outgoingDoc["error"]["message"] = errorMessage;

outgoingDoc["error"]["data"]["method"] = methodName;

outgoingDoc["id"] = id;

// Send response

}

After making these changes, your Arduino code should support method-based routing for JSON-RPC requests.

What should go here:

else {

      // Handle deserialization error

    }

In the else block, you should handle the case where the JSON deserialization fails, which means the incoming JSON data is malformed or cannot be parsed correctly. Here's an example of how you can handle this situation:

else {

// Handle deserialization error

respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]);

Serial.print("deserializeJson() failed: ");

Serial.println(error.c\_str());

}

In this example:

1. **Respond with an error**: Call respondWithError() to send a JSON-RPC error response back to the client, indicating that the JSON deserialization failed. The error message includes the text "JSON deserialization failed" and the actual error description from the error.c\_str().
2. **Print the error for debugging**: Use Serial.print() and Serial.println() to print the error message to the serial console for debugging purposes. This helps you understand what went wrong during deserialization.

**Updated respondWithError() function to accommodate error codes and messages:**

To make the error handling more informative, consider updating the respondWithError() function to accept an error code and message:

void respondWithError(const char\* errorMessage, const char\* errorDetails, int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["error"]["code"] = -32600; // Parse error (you can customize this)

outgoingDoc["error"]["message"] = errorMessage;

outgoingDoc["error"]["data"]["details"] = errorDetails;

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Error Response:");

Serial.println(outgoingJson);

}

This updated function includes an error details section in the response, providing more context about the error.

* **I had this code segment before. Is it still necessary?**

if (error) {

        Serial.print("deserializeJson() failed: ");

        Serial.println(error.f\_str());

        respondWithError("Parse error", 1); // Assuming id is always present for simplicity

      } else {

        if (incomingDoc["method"] == "getSensorReadings") {

          const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device

          if (strcmp(targetDevice, LOCAL\_NAME) == 0) { // If request is for this device

            respondWithSensorReadings(incomingDoc["id"]);

          } else { // Ignore requests not meant for this device

            Serial.println("Ignoring request not meant for this device.");

          }

        } else {

          respondWithError("Method not found", incomingDoc["id"]);

        }

With the new code segment I provided earlier, the old one is partially redundant. Here's a breakdown of what's changed and what you can keep or remove:

**Redundant parts:**

1. **Error handling**: The new code segment already handles deserialization errors with a more informative error response:

else { // Handle deserialization error respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]); Serial.print("deserializeJson() failed: "); Serial.println(error.c\_str()); }

You can remove the old error handling block:

```cpp

if (error) {

Serial.print("deserializeJson() failed: ");

Serial.println(error.f\_str());

respondWithError("Parse error", 1); // Assuming id is always present for simplicity

}

1. **Method not found error**: The new code segment also handles unknown methods with a more informative error response:

else { respondWithError("Unknown method", incomingDoc["method"], incomingDoc["id"]); }

You can remove the old "Method not found" error handling block:

```cpp

else {

respondWithError("Method not found", incomingDoc["id"]);

}

**Parts you might want to keep or modify:**

1. **Device targeting**: The old code segment checks if the incoming request is targeted at the current device:

const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device if (strcmp(targetDevice, LOCAL\_NAME) == 0) { // If request is for this device respondWithSensorReadings(incomingDoc["id"]); } else { // Ignore requests not meant for this device Serial.println("Ignoring request not meant for this device."); }

If you still want to implement device targeting, you can integrate this logic into the new code segment. For example:

```cpp

if (strcmp(method, "getSensorReadings") == 0) {

const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device

if (strcmp(targetDevice, LOCAL\_NAME) == 0) { // If request is for this device

respondWithSensorReadings(incomingDoc["id"]);

} else { // Ignore requests not meant for this device

Serial.println("Ignoring request not meant for this device.");

}

}

**Updated code segment with device targeting:**

Here's the refactored code segment that combines the new error handling with device targeting for the "getSensorReadings" method:

if (jsonRpcCharacteristic.written()) {

// ...

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (!error) {

const char\* method = incomingDoc["method"];

if (strcmp(method, "getTemperature") == 0) {

respondWithTemperature(incomingDoc["id"]);

} else if (strcmp(method, "getHumidity") == 0) {

respondWithHumidity(incomingDoc["id"]);

} else if (strcmp(method, "getSensorReadings") == 0) {

const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device

if (strcmp(targetDevice, LOCAL\_NAME) == 0) { // If request is for this device

respondWithSensorReadings(incomingDoc["id"]);

} else { // Ignore requests not meant for this device

Serial.println("Ignoring request not meant for this device.");

}

} else {

respondWithError("Unknown method", method, incomingDoc["id"]);

}

} else {

// Handle deserialization error

respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]);

Serial.print("deserializeJson() failed: ");

Serial.println(error.c\_str());

}

}

* **But code should also check if request is for that peripheral device when it get getTemperature or getHumidity methods**

To ensure that the peripheral device only responds to requests intended for it, you should indeed check the target device for all methods, not just getSensorReadings. Here's the updated code segment:

if (jsonRpcCharacteristic.written()) {

// ...

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (!error) {

const char\* method = incomingDoc["method"];

const char\* targetDevice = incomingDoc["params"][0]["device"]; // Check target device

if (strcmp(targetDevice, LOCAL\_NAME) != 0) { // If request is not for this device

Serial.println("Ignoring request not meant for this device.");

return; // Ignore requests not meant for this device

}

if (strcmp(method, "getTemperature") == 0) {

respondWithTemperature(incomingDoc["id"]);

} else if (strcmp(method, "getHumidity") == 0) {

respondWithHumidity(incomingDoc["id"]);

} else if (strcmp(method, "getSensorReadings") == 0) {

respondWithSensorReadings(incomingDoc["id"]);

} else {

respondWithError("Unknown method", method, incomingDoc["id"]);

}

} else {

// Handle deserialization error

respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]);

Serial.print("deserializeJson() failed: ");

Serial.println(error.c\_str());

}

}

However, there's a small issue with this updated code: not all methods might have a targetDevice parameter in the paramsarray. To handle this, you should add a check to ensure that targetDevice exists before comparing it with LOCAL\_NAME. Here's the refactored code:

if (jsonRpcCharacteristic.written()) {

// ...

DynamicJsonDocument incomingDoc(2048);

DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

if (!error) {

const char\* method = incomingDoc["method"];

// Check if targetDevice exists in params

const char\* targetDevice = NULL;

if (incomingDoc["params"].is<JsonArray>() && incomingDoc["params"].size() > 0) {

JsonVariant params = incomingDoc["params"][0];

if (params.is<JsonObject>() && params.as<JsonObject>().containsKey("device")) {

targetDevice = params["device"];

}

}

// Check if request is for this device

if (targetDevice != NULL && strcmp(targetDevice, LOCAL\_NAME) != 0) { // If request is not for this device

Serial.println("Ignoring request not meant for this device.");

return; // Ignore requests not meant for this device

}

if (strcmp(method, "getTemperature") == 0) {

respondWithTemperature(incomingDoc["id"]);

} else if (strcmp(method, "getHumidity") == 0) {

respondWithHumidity(incomingDoc["id"]);

} else if (strcmp(method, "getSensorReadings") == 0) {

respondWithSensorReadings(incomingDoc["id"]);

} else {

respondWithError("Unknown method", method, incomingDoc["id"]);

}

} else {

// Handle deserialization error

respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]);

Serial.print("deserializeJson() failed: ");

Serial.println(error.c\_str());

}

}

This refactored code checks if targetDevice exists in the paramsarray before comparing it with LOCAL\_NAME, ensuring that the code doesn't crash when encountering methods without a targetDevice parameter.

* **I dont understand why this is necessary:**

// Check if targetDevice exists in params

    const char\* targetDevice = NULL;

    if (incomingDoc["params"].is<JsonArray>() && incomingDoc["params"].size() > 0) {

      JsonVariant params = incomingDoc["params"][0];

      if (params.is<JsonObject>() && params.as<JsonObject>().containsKey("device")) {

        targetDevice = params["device"];

      }

    }

Let's break down the necessity of this code snippet:

**What it does:**

This code checks if a targetDevice parameter exists within the params array of the incoming JSON document. If it does, the code extracts the targetDevice value.

**Why it's necessary:**

1. **Avoid Null Pointer Dereferences**: Without this check, if targetDevice is not present in the params array, attempting to access incomingDoc["params"][0]["device"] would result in a null pointer dereference. This could lead to a crash or undefined behavior.
2. **Handle Variable JSON Structures**: The incoming JSON document might have varying structures depending on the method or the sender. This check ensures the code can handle different JSON structures without failing.
3. **Prevent Key Not Found Errors**: If the device key is not present within the params array, this check prevents the code from attempting to access a non-existent key, which would otherwise result in an error.

**Simplified Example to Illustrate the Issue:**

Suppose you have two possible JSON requests:

**Request 1 (with targetDevice):**

{

"jsonrpc": "2.0",

"method": "getTemperature",

"params": [

{

"device": "Nano33BLE-JSON-RPC-1"

}

],

"id": 1

}

**Request 2 (without targetDevice):**

{

"jsonrpc": "2.0",

"method": "getTemperature",

"params": [],

"id": 1

}

Without the existence check, if the code receives **Request 2**, attempting to access incomingDoc["params"][0]["device"] would fail because params is an empty array, and there is no device key to access.

**Alternative Approach:**

If you're certain that all incoming requests will always have a targetDevice parameter in the params array, you could simplify the code by assuming its existence. However, this approach is more prone to errors if the JSON structure varies:

const char\* targetDevice = incomingDoc["params"][0]["device"];

Keep in mind that robust code should always validate and handle potential errors to ensure reliability and prevent crashes.

* **So now this is correct arduino code, right?**

#include <ArduinoBLE.h>

#include <ArduinoJson.h>

// **CHANGE THIS FOR EACH DEVICE**

const char\* LOCAL\_NAME = "Nano33BLE-JSON-RPC-1"; // Use "Nano33BLE-JSON-RPC-2" for the second device

// Simulated sensor readings for demo purposes

float readTemperature() { return 22.5; } // Replace with actual sensor read function

float readHumidity()    { return 60.2; } // Replace with actual sensor read function

BLEService jsonRpcService("91ED0001-0000-0000-0000-000000000000");

BLECharacteristic jsonRpcCharacteristic("91ED0002-0000-0000-0000-000000000000", BLERead | BLEWrite, "json-rpc");

void setup() {

  Serial.begin(9600);

  while (!Serial);

  if (!BLE.begin()) {

    Serial.println("Starting BLE failed!");

    while (1);

  }

  BLE.setLocalName(LOCAL\_NAME);

  BLE.setAdvertisedService(jsonRpcService);

  jsonRpcService.addCharacteristic(jsonRpcCharacteristic);

  BLE.addService(jsonRpcService);

  jsonRpcCharacteristic.writeValue("");

  BLE.advertise();

  Serial.println("BLE JSON-RPC Service Started. Waiting for connections...");

}

void loop() {

  BLE.poll();

  if (BLE.connected()) {

    if (jsonRpcCharacteristic.written()) {

      size\_t jsonDataLength = jsonRpcCharacteristic.valueLength();

      const char\* jsonData = (const char\*)jsonRpcCharacteristic.value();

      String incomingJson = String(jsonData, jsonDataLength);

      Serial.println("Incoming JSON-RPC Request:");

      Serial.println(incomingJson);

      DynamicJsonDocument incomingDoc(2048);

      DeserializationError error = deserializeJson(incomingDoc, jsonData, jsonDataLength);

      if (!error) {

        const char\* method = incomingDoc["method"];

        // Check if targetDevice exists in params

        const char\* targetDevice = NULL;

        if (incomingDoc["params"].is<JsonArray>() && incomingDoc["params"].size() > 0) {

          JsonVariant params = incomingDoc["params"][0];

          if (params.is<JsonObject>() && params.as<JsonObject>().containsKey("device")) {

            targetDevice = params["device"];

          }

        }

        // Check if request is for this device

        if (targetDevice != NULL && strcmp(targetDevice, LOCAL\_NAME) != 0) { // If request is not for this device

          Serial.println("Ignoring request not meant for this device.");

          return; // Ignore requests not meant for this device

        }

        if (strcmp(method, "getTemperature") == 0) {

          respondWithTemperature(incomingDoc["id"]);

        } else if (strcmp(method, "getHumidity") == 0) {

          respondWithHumidity(incomingDoc["id"]);

        } else if (strcmp(method, "getSensorReadings") == 0) {

          respondWithSensorReadings(incomingDoc["id"]);

        } else {

          respondWithError("Unknown method", method, incomingDoc["id"]);

        }

      } else {

        // Handle deserialization error

        respondWithError("JSON deserialization failed", error.c\_str(), incomingDoc["id"]);

        Serial.print("deserializeJson() failed: ");

        Serial.println(error.c\_str());

      }

    }

  }

}

void respondWithError(const char\* errorMessage, const char\* methodName, int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["error"]["code"] = -32601; // Method not found

  outgoingDoc["error"]["message"] = errorMessage;

  outgoingDoc["error"]["data"]["method"] = methodName;

  outgoingDoc["id"] = id;

  // Send response

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

  Serial.println("Sent Error Response:");

  Serial.println(outgoingJson);

}

// OTPRIJE

// void respondWithError(const char\* errorMessage, int id) {

//   DynamicJsonDocument outgoingDoc(2048);

//   outgoingDoc["jsonrpc"] = "2.0";

//   outgoingDoc["error"]["code"] = -32600; // Method not found

//   outgoingDoc["error"]["message"] = errorMessage;

//   outgoingDoc["id"] = id;

//   String outgoingJson;

//   serializeJson(outgoingDoc, outgoingJson);

//   jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

//   Serial.println("Sent Error Response:");

//   Serial.println(outgoingJson);

// }

// Example response functions

void respondWithTemperature(int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["result"]["temperature"] = readTemperature();

  outgoingDoc["id"] = id;

  // Send response

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

  Serial.println("Sent Temperature Readings Response:");

  Serial.println(outgoingJson);

}

void respondWithHumidity(int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["result"]["humidity"] = readHumidity();

  outgoingDoc["id"] = id;

  // Send response

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

  Serial.println("Sent Humidity Readings Response:");

  Serial.println(outgoingJson);

}

void respondWithSensorReadings(int id) {

  DynamicJsonDocument outgoingDoc(2048);

  outgoingDoc["jsonrpc"] = "2.0";

  outgoingDoc["result"]["temperature"] = readTemperature();

  outgoingDoc["result"]["humidity"] = readHumidity();

  outgoingDoc["id"] = id;

  String outgoingJson;

  serializeJson(outgoingDoc, outgoingJson);

  jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

  Serial.println("Sent Sensor Readings Response:");

  Serial.println(outgoingJson);

}

Your Arduino code looks mostly correct, but there are a few missing pieces and minor adjustments needed for it to work seamlessly with the JSON-RPC protocol and the laptop code. Here's a breakdown of the necessary changes:

**1. Missing respondWithError function for deserialization error:**

* You have two respondWithError functions, but the one for deserialization errors is commented out. Uncomment the first respondWithError function or modify the second one to accommodate the error message and code.

**2. Inconsistent error codes:**

* In the respondWithError function, you're using error code -32601for both unknown methods and deserialization errors. Consider using distinct error codes for clarity, such as -32600 for method not found and -32700 for parse error (deserialization).

**3. No validation for id in response functions:**

* Although not strictly necessary, it's a good practice to validate the id parameter in your response functions to ensure it's not null or empty.

**4. Response functions' comments:**

* The comments in your response functions (// Send response) are accurate but can be improved for clarity. Consider rephrasing them to indicate that the response is being sent back to the client.

**Updated Code Snippets:**

**respondWithError function for deserialization error:**

void respondWithError(const char\* errorMessage, const char\* errorDetails, int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["error"]["code"] = -32700; // Parse error

outgoingDoc["error"]["message"] = errorMessage;

outgoingDoc["error"]["data"]["details"] = errorDetails;

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Error Response (Deserialization):");

Serial.println(outgoingJson);

}

**Updated respondWithError function for unknown method:**

void respondWithError(const char\* errorMessage, const char\* methodName, int id) {

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["error"]["code"] = -32600; // Method not found

outgoingDoc["error"]["message"] = errorMessage;

outgoingDoc["error"]["data"]["method"] = methodName;

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Error Response (Unknown Method):");

Serial.println(outgoingJson);

}

**Validation for id in response functions:**

void respondWithTemperature(int id) {

if (id == 0) {

Serial.println("Warning: ID is null or empty.");

return;

}

DynamicJsonDocument outgoingDoc(2048);

outgoingDoc["jsonrpc"] = "2.0";

outgoingDoc["result"]["temperature"] = readTemperature();

outgoingDoc["id"] = id;

String outgoingJson;

serializeJson(outgoingDoc, outgoingJson);

jsonRpcCharacteristic.writeValue((const uint8\_t\*)outgoingJson.c\_str(), outgoingJson.length());

Serial.println("Sent Temperature Readings Response to Client:");

Serial.println(outgoingJson);

}

Apply these changes to your Arduino code to ensure it's robust and consistent with the JSON-RPC protocol.

* **Could you rewrite the following python code to work with new ardunio code?**

import asyncio

from bleak import BleakClient, BleakScanner

import json

**Service and Characteristic UUIDs (Match those in the Arduino code)**

JSON\_RPC\_SERVICE\_UUID = "91ED0001-0000-0000-0000-000000000000"

JSON\_RPC\_CHARACTERISTIC\_UUID = "91ED0002-0000-0000-0000-000000000000"

**Arduino Nano 33 BLE Device Names**

TARGET\_DEVICE\_NAMES = ["Nano33BLE-JSON-RPC-1", "Nano33BLE-JSON-RPC-2"]

async def main():

    # Scan for devices

    print("Scanning for devices...")

    scanner = BleakScanner()

    await scanner.start()

    await asyncio.sleep(5)  # Scan for 5 seconds

    await scanner.stop()

    # Find target devices

    target\_devices = {device.name: device for device in scanner.discovered\_devices if device.name in TARGET\_DEVICE\_NAMES}

    if not target\_devices:

        print("No target devices found.")

        return

    # Select target device

    print("Select target device:")

    for i, (name, device) in enumerate(target\_devices.items()):

        print(f"{i+1}. {name} - {device.address}")

    choice = input("Enter the number of your choice: ")

    try:

        choice = int(choice)

        if choice < 1 or choice > len(target\_devices):

            print("Invalid choice.")

            return

    except ValueError:

        print("Invalid input. Please enter a number.")

        return

    selected\_name = list(target\_devices.keys())[choice - 1]

    target\_device = target\_devices[selected\_name]

    print(f"Connecting to {selected\_name} - {target\_device.address}...")

    async with BleakClient(target\_device) as client:

        print(f"Connected to {selected\_name}")

        # Discover the service and characteristic

        svc = await client.get\_service(JSON\_RPC\_SERVICE\_UUID)

        if not svc:

            print(f"Failed to find service {JSON\_RPC\_SERVICE\_UUID}")

            return

        char = next((c for c in svc.characteristics if c.uuid == JSON\_RPC\_CHARACTERISTIC\_UUID), None)

        if not char:

            print(f"Failed to find characteristic {JSON\_RPC\_CHARACTERISTIC\_UUID} in service")

            return

        # Prepare and send the JSON-RPC request

        request\_id = 1

        target\_device\_name = selected\_name  # Use the selected device's name in the request

        json\_rpc\_request = {

            "jsonrpc": "2.0",

            "method": "getSensorReadings",

            "params": [{"device": target\_device\_name}],  # Specify the target device

            "id": request\_id

        }

        request\_json = json.dumps(json\_rpc\_request)

        print(f"Sending JSON-RPC Request: {request\_json}")

        await client.write\_gatt\_char(char, request\_json.encode())

        # Wait for the response (Note: This example assumes the response comes immediately after the request.

        #  In a real-world scenario, consider implementing a more robust waiting mechanism.)

        await asyncio.sleep(1)  # Wait for 1 second

        # Read the response (Assuming it's available in the characteristic value)

        response = await client.read\_gatt\_char(char)

        response\_json = response.decode()

        print(f"Received JSON-RPC Response: {response\_json}")

        try:

            response\_data = json.loads(response\_json)

            if "result" in response\_data:

                print(f"Temperature: {response\_data['result']['temperature']}°C, Humidity: {response\_data['result']['humidity']}%")

            else:

                print("No sensor readings in response.")

        except json.JSONDecodeError as e:

            print(f"Failed to parse JSON response: {e}")

asyncio.run(main())

To make the provided Python code work with the new Arduino code, you'll need to modify the JSON-RPC requests to accommodate the changes in the Arduino code. Here's the updated Python code:

import asyncio

from bleak import BleakClient, BleakScanner

import json

# Service and Characteristic UUIDs (Match those in the Arduino code)

JSON\_RPC\_SERVICE\_UUID = "91ED0001-0000-0000-0000-000000000000"

JSON\_RPC\_CHARACTERISTIC\_UUID = "91ED0002-0000-0000-0000-000000000000"

# Arduino Nano 33 BLE Device Names

TARGET\_DEVICE\_NAMES = ["Nano33BLE-JSON-RPC-1", "Nano33BLE-JSON-RPC-2"]

async def main():

# Scan for devices

print("Scanning for devices...")

scanner = BleakScanner()

await scanner.start()

await asyncio.sleep(5) # Scan for 5 seconds

await scanner.stop()

# Find target devices

target\_devices = {device.name: device for device in scanner.discovered\_devices if device.name in TARGET\_DEVICE\_NAMES}

if not target\_devices:

print("No target devices found.")

return

# Select target device

print("Select target device:")

for i, (name, device) in enumerate(target\_devices.items()):

print(f"{i+1}. {name} - {device.address}")

choice = input("Enter the number of your choice: ")

try:

choice = int(choice)

if choice < 1 or choice > len(target\_devices):

print("Invalid choice.")

return

except ValueError:

print("Invalid input. Please enter a number.")

return

selected\_name = list(target\_devices.keys())[choice - 1]

target\_device = target\_devices[selected\_name]

print(f"Connecting to {selected\_name} - {target\_device.address}...")

async with BleakClient(target\_device) as client:

print(f"Connected to {selected\_name}")

# Discover the service and characteristic

svc = await client.get\_service(JSON\_RPC\_SERVICE\_UUID)

if not svc:

print(f"Failed to find service {JSON\_RPC\_SERVICE\_UUID}")

return

char = next((c for c in svc.characteristics if c.uuid == JSON\_RPC\_CHARACTERISTIC\_UUID), None)

if not char:

print(f"Failed to find characteristic {JSON\_RPC\_CHARACTERISTIC\_UUID} in service")

return

# Prepare and send JSON-RPC requests for different methods

methods = ["getTemperature", "getHumidity", "getSensorReadings"]

for method in methods:

request\_id = 1

target\_device\_name = selected\_name # Use the selected device's name in the request

json\_rpc\_request = {

"jsonrpc": "2.0",

"method": method,

"params": [{"device": target\_device\_name}], # Specify the target device

"id": request\_id

}

request\_json = json.dumps(json\_rpc\_request)

print(f"Sending JSON-RPC Request for {method}: {request\_json}")

await client.write\_gatt\_char(char, request\_json.encode())

# Wait for the response (Note: This example assumes the response comes immediately after the request.

# In a real-world scenario, consider implementing a more robust waiting mechanism.)

await asyncio.sleep(1) # Wait for 1 second

# Read the response (Assuming it's available in the characteristic value)

response = await client.read\_gatt\_char(char)

response\_json = response.decode()

print(f"Received JSON-RPC Response for {method}: {response\_json}")

try:

response\_data = json.loads(response\_json)

if "result" in response\_data:

if method == "getSensorReadings":

print(f"Temperature: {response\_data['result']['temperature']}°C, Humidity: {response\_data['result']['humidity']}%")

elif method == "getTemperature":

print(f"Temperature: {response\_data['result']['temperature']}°C")

elif method == "getHumidity":

print(f"Humidity: {response\_data['result']['humidity']}%")

else:

print("No sensor readings in response.")

except json.JSONDecodeError as e:

print(f"Failed to parse JSON response: {e}")

asyncio.run(main())

**Changes Made:**

1. **Added Support for Multiple Methods**: The updated code sends JSON-RPC requests for getTemperature, getHumidity, and getSensorReadings methods.
2. **Modified Response Handling**: The code now handles responses for each method, printing the corresponding sensor readings.