BENOIT BLANCHON
CREATOR OF ARDUINOJSON

Mastering ArduinoJson 7

Efficient JSON serialization for embedded C++

Contents

Contents **iv**

1 Introduction **1**
1.1 About this book . . . 2
1.1.1 Overview . . . 2
1.1.2 Code samples . . . 2
1.1.3 What changed since Mastering ArduinoJson 6 . . . 3
1.2 Introduction to JSON . . . 4
1.2.1 What is JSON? . . . 4
1.2.2 What is serialization? . . . 5
1.2.3 What can you do with JSON? . . . 5
1.2.4 History of JSON . . . 8
1.2.5 Why is JSON so popular? . . . 9
1.2.6 The JSON syntax . . . 9
1.2.7 Binary data in JSON . . . 13
1.2.8 Comments in JSON . . . 13
1.3 Introduction to ArduinoJson . . . 15
1.3.1 What ArduinoJson is . . . 15
1.3.2 What ArduinoJson is not . . . 15
1.3.3 What makes ArduinoJson different? . . . 16
1.3.4 Does size matter? . . . 18
1.3.5 What are the alternatives to ArduinoJson? . . . 18
1.3.6 How to install ArduinoJson . . . 20
1.3.7 The examples . . . 25
1.4 Summary . . . 27

2 The missing C++ course **28**
2.1 Why a C++ course? . . . 29
2.2 Harvard and von Neumann architectures . . . 31
2.3 Stack, heap, and globals . . . 33
2.3.1 Globals . . . 34

2.3.2 Heap 35
2.3.3 Stack 36
2.4 Pointers 38
2.4.1 What is a pointer? 38
2.4.2 Dereferencing a pointer 38
2.4.3 Pointers and arrays 39
2.4.4 Taking the address of a variable 40
2.4.5 Pointer to `class` and `struct` 40
2.4.6 Pointer to constant 41
2.4.7 The null pointer 43
2.4.8 Why use pointers? 44
2.5 Memory management 45
2.5.1 `malloc()` and `free()` 45
2.5.2 `new` and `delete` 45
2.5.3 Smart pointers 46
2.5.4 RAII 48
2.6 References 49
2.6.1 What is a reference? 49
2.6.2 Differences with pointers 49
2.6.3 Reference to constant 50
2.6.4 Rules of references 51
2.6.5 Common problems 51
2.6.6 Usage for references 52
2.7 Strings 53
2.7.1 How are the strings stored? 53
2.7.2 String literals in RAM 53
2.7.3 String literals in Flash 54
2.7.4 Pointer to the “globals” section 55
2.7.5 Mutable string in “globals” 56
2.7.6 A copy in the stack 57
2.7.7 A copy in the heap 58
2.7.8 A word about the `String` class 59
2.7.9 Passing strings to functions 60
2.8 Summary 62

3 Deserialize with ArduinoJson 64
3.1 The example of this chapter 65
3.2 Deserializing an object 66
3.2.1 The JSON document 66
3.2.2 Deserializing the JSON document 66

3.3 Extracting values from an object . . . 68
3.3.1 Extracting values . . . 68
3.3.2 Explicit casts . . . 68
3.3.3 When values are missing . . . 69
3.3.4 Changing the default value . . . 70
3.4 Inspecting an unknown object . . . 71
3.4.1 Getting a reference to the object . . . 71
3.4.2 Enumerating the keys . . . 72
3.4.3 Detecting the type of value . . . 72
3.4.4 Variant types and C++ types . . . 73
3.4.5 Testing if a key exists in an object . . . 73
3.5 Deserializing an array . . . 75
3.5.1 The JSON document . . . 75
3.5.2 Parsing the array . . . 75
3.6 Extracting values from an array . . . 77
3.6.1 Retrieving elements by index . . . 77
3.6.2 Alternative syntaxes . . . 77
3.6.3 When complex values are missing . . . 78
3.7 Inspecting an unknown array . . . 80
3.7.1 Getting a reference to the array . . . 80
3.7.2 Number of elements in an array . . . 80
3.7.3 Iteration . . . 81
3.7.4 Detecting the type of an element . . . 81
3.8 Reading from a stream . . . 83
3.8.1 Reading from a file . . . 83
3.8.2 Reading from an HTTP response . . . 84
3.9 The ArduinoJson Assistant . . . 92
3.9.1 Step 1: Configuration . . . 93
3.9.2 Step 2: JSON . . . 94
3.9.3 Step 3: Program . . . 95
3.10 Summary . . . 96

4 Serializing with ArduinoJson 98
4.1 The example of this chapter . . . 99
4.2 Creating an object . . . 100
4.2.1 The example . . . 100
4.2.2 Creating the JsonDocument . . . 100
4.2.3 Adding members . . . 101
4.2.4 Creating an empty object . . . 101
4.2.5 Replacing and removing members . . . 102

4.3 Creating an array 103
4.3.1 The example 103
4.3.2 Adding elements 103
4.3.3 Adding nested objects 104
4.3.4 Creating an empty array 105
4.3.5 Replacing and removing elements 105
4.4 Writing to memory 106
4.4.1 Minified JSON 106
4.4.2 Specifying (or not) the buffer size 106
4.4.3 Prettified JSON 107
4.4.4 Measuring the length 108
4.4.5 Writing to a `String` 109
4.4.6 Casting a `JsonVariant` to a `String` 109
4.5 Writing to a stream 110
4.5.1 What's an output stream? 110
4.5.2 Writing to the serial port 111
4.5.3 Writing to a file 112
4.5.4 Writing to a TCP connection 113
4.6 Duplication of strings 118
4.6.1 An example 118
4.6.2 Keys and values 119
4.6.3 Copy only occurs when adding values 119
4.7 Inserting special values 120
4.7.1 Adding `null` 120
4.7.2 Adding pre-formatted JSON 120
4.8 The ArduinoJson Assistant 122
4.8.1 Step 1: Configuration 122
4.8.2 Step 2: JSON 123
4.8.3 Step 3: Program 123
4.9 Summary 125

5 Advanced Techniques **127**
5.1 Introduction 128
5.2 Filtering the input 129
5.3 Deserializing in chunks 134
5.4 JSON streaming 139
5.5 Using external RAM 142
5.6 Logging 145
5.7 Buffering 148
5.8 Custom readers and writers 151

5.9 Custom converters . . . 156
5.10 MessagePack . . . 162
5.11 ArduinoJson Assistant's Tweaks . . . 165
5.11.1 Floating-point storage . . . 166
5.11.2 Integer storage . . . 166
5.11.3 String deduplication . . . 167
5.11.4 `const char*` strings . . . 168
5.12 Summary . . . 169

6 Inside ArduinoJson **171**
6.1 Introduction . . . 172
6.2 Variants . . . 173
6.3 Integers . . . 175
6.4 String . . . 177
6.4.1 String nodes . . . 177
6.4.2 String adapters . . . 178
6.4.3 Variable-length arrays . . . 178
6.5 Arrays and objects . . . 180
6.6 Document tree . . . 182
6.7 Slot pool . . . 184
6.8 The resource manager . . . 186
6.9 Smart pointers . . . 187
6.9.1 `JsonVariant` . . . 187
6.9.2 Unbound `JsonVariant` . . . 187
6.9.3 `JsonVariantConst` . . . 188
6.9.4 `JsonArray` and `JsonObject` . . . 189
6.10 Comparison operators . . . 190
6.11 Converters . . . 191
6.12 Proxies . . . 193
6.13 Deserializers . . . 195
6.13.1 Reading from various types . . . 195
6.13.2 Reading one character at a time . . . 196
6.13.3 Nesting limit . . . 196
6.13.4 Escape sequences . . . 198
6.13.5 Filtering . . . 199
6.13.6 String-to-float conversion . . . 199
6.14 Serializers . . . 200
6.14.1 Writing to various types . . . 200
6.14.2 Float-to-string conversion . . . 201
6.15 Namespaces . . . 202

6.16 Aggregated header . . . 203
6.17 Summary . . . 204

7 Troubleshooting **206**
7.1 Introduction . . . 207
7.2 Program crashes . . . 208
7.2.1 Undefined Behaviors . . . 208
7.2.2 A bug in ArduinoJson? . . . 208
7.2.3 Null string . . . 209
7.2.4 Use after free . . . 209
7.2.5 Return of stack variable address . . . 211
7.2.6 Buffer overflow . . . 212
7.2.7 Stack overflow . . . 213
7.2.8 How to diagnose these bugs . . . 214
7.2.9 How to prevent these bugs? . . . 216
7.3 Deserialization issues . . . 219
7.3.1 `EmptyInput` . . . 219
7.3.2 `IncompleteInput` . . . 220
7.3.3 `InvalidInput` . . . 222
7.3.4 `NoMemory` . . . 226
7.3.5 `TooDeep` . . . 226
7.4 Serialization issues . . . 228
7.4.1 The JSON document is incomplete . . . 228
7.4.2 The JSON document contains garbage . . . 228
7.4.3 The serialization is slow . . . 229
7.5 Common error messages . . . 231
7.5.1 Invalid conversion from `const char*` to `char*` . . . 231
7.5.2 Invalid conversion from `const char*` to `int` . . . 231
7.5.3 No match for `operator[]` . . . 232
7.5.4 Ambiguous overload for `operator=` . . . 233
7.5.5 Call of overloaded function is ambiguous . . . 234
7.6 Asking for help . . . 235
7.7 Summary . . . 237

8 Case Studies **238**
8.1 JSON Configuration File . . . 239
8.1.1 Presentation . . . 239
8.1.2 The JSON document . . . 239
8.1.3 The configuration class . . . 240
8.1.4 Converters . . . 241

8.1.5 Saving the configuration to a file . . . 244
8.1.6 Reading the configuration from a file . . . 245
8.1.7 Conclusion . . . 245
8.2 OpenWeatherMap on MKR1000 . . . 247
8.2.1 Presentation . . . 247
8.2.2 OpenWeatherMap's API . . . 247
8.2.3 The JSON response . . . 248
8.2.4 Reducing the size of the document . . . 250
8.2.5 The filter document . . . 252
8.2.6 The code . . . 253
8.2.7 Summary . . . 254
8.3 Reddit on ESP8266 . . . 255
8.3.1 Presentation . . . 255
8.3.2 Reddit's API . . . 256
8.3.3 The response . . . 257
8.3.4 The main loop . . . 258
8.3.5 Sending the request . . . 259
8.3.6 Assembling the puzzle . . . 259
8.3.7 Summary . . . 260
8.4 RESTful client . . . 262
8.4.1 Presentation . . . 262
8.4.2 JSON-RPC Request . . . 263
8.4.3 JSON-RPC Response . . . 263
8.4.4 A reusable RESTful client . . . 264
8.4.5 Sending notification to Kodi . . . 268
8.4.6 Reading Kodi's version . . . 270
8.4.7 Summary . . . 272
8.5 Recursive analyzer . . . 273
8.5.1 Presentation . . . 273
8.5.2 Reading from the serial port . . . 274
8.5.3 Flushing after an error . . . 275
8.5.4 Testing the type of a `JsonVariant` . . . 275
8.5.5 Printing values . . . 277
8.5.6 Summary . . . 279

9 Conclusion 280

Index 281

Chapter 4

Serializing with ArduinoJson

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*
>
> – **Martin Fowler, Refactoring: Improving the Design of Existing Code**

4.1 The example of this chapter

Reading a JSON document is only half the story; we'll now see how to write a JSON document with ArduinoJson.

In the previous chapter, we played with GitHub's API. We'll use a example for this chapter: pushing data to Adafruit IO.

Adafruit IO is a cloud storage service for IoT data: it collects the data coming from your devices and can trigger some action, such as sending an email when a criteria is met.

They offer a free plan with the following restrictions:

- 30 data points per minute
- 30 days of data storage
- 10 feeds

If you need more, it's just $10 a month.

The service is very easy to use. All you need is an Adafruit account (yes, you can use the account from the Adafruit shop).

As we did in the previous chapter, we'll start with a simple JSON document and add complexity step by step.

Since Adafruit IO doesn't impose a secure connection, we can use a less powerful microcontroller than in the previous chapter; we'll use an Arduino UNO with an Ethernet Shield.

4.2 Creating an object

4.2.1 The example

Here is the JSON object we want to create:

```json
{
  "value": 42,
  "lat": 48.748010,
  "lon": 2.293491
}
```

It's a flat object, meaning that it has no nested object or array, and it contains the following members:

1. `"value"` is the integer we want to save in Adafruit IO.
2. `"lat"` is the latitude coordinate.
3. `"lon"` is the longitude coordinate.

Adafruit IO supports other optional members (like the elevation coordinate and the measurement time), but the three members above are sufficient for our example.

4.2.2 Creating the JsonDocument

As for the deserialization, we start by creating a `JsonDocument` to hold the memory representation of the object. The previous chapter introduced `JsonDocument`, so I assume you're familiar with it.

```
JsonDocument doc;
```

The `JsonDocument` is currently empty, and `JsonDocument::isNull()` returns `true`. If we serialized it now, the output would be "`null`."

4.2.3 Adding members

An empty `JsonDocument` automatically becomes an object when we add members to it. We do that with the subscript operator (`[]`), just like we did in the previous chapter:

```
doc["value"] = 42;
doc["lat"] = 48.748010;
doc["lon"] = 2.293491;
```

If there is not enough memory to store a new value in the `JsonDocument`, the new value is silently ignored. However, you can detect if some values are missing by checking `JsonDocument::overflowed()`, which returns `true` when an allocation failed.

To be honest, I never check `JsonDocument::overflowed()` in my programs. The reason is simple: the JSON document is roughly the same for each iteration; if it works once, it always works. There is no reason to bloat the code for a situation that cannot happen.

4.2.4 Creating an empty object

We just saw that the `JsonDocument` becomes an object as soon as you insert a member, but what if you don't have any members to add? What if you want to create an empty object?

When you need an empty object, you can no longer rely on the implicit conversion. Instead, you must explicitly convert the `JsonDocument` to a `JsonObject` with `JsonDocument::to<JsonObject>()`:

```
// Convert the document to an object
JsonObject obj = doc.to<JsonObject>();
```

This function clears the `JsonDocument`, so all existing references become invalid. Then, it creates an empty object at the root of the document and returns a reference to this object.

At this point, the `JsonDocument` is not empty anymore, and `JsonDocument::isNull()` returns `false`. If we serialized this document, the output would be "`{}`".

4.2.5 Replacing and removing members

Naturally, it's possible to replace a member in the object, for example:

```
obj["value"] = 42;
obj["value"] = 43;
```

Finally, you can remove a member with `JsonObject::remove(key)`, for example:

```
// Remove the "value" field
obj.remove("value ");
```

4.3 Creating an array

4.3.1 The example

Now that we can create objects, let's see how to create an array. Our new example will be an array that contains two objects.

```
[
  {
    "key": "a1",
    "value": 12
  },
  {
    "key": "a2",
    "value": 34
  }
]
```

The values `12` and `34` are just placeholder; in reality, we'll use the result from `analogRead()`.

4.3.2 Adding elements

In the previous section, we saw that an empty `JsonDocument` automatically becomes an object as soon as we insert the first member. This statement was only partially correct: it becomes an object as soon as we use it as an object.

Indeed, if we treat an empty `JsonDocument` as an array, it automatically becomes an array. For example, this happens if we call `JsonDocument::add()` like so:

```
JsonDocument doc;
doc.add(1);
doc.add(2);
```

After these two lines, the `JsonDocument` contains `[1,2]`.

Alternatively, we can create the same array with the `[]` operator like so:

```
doc[0] = 1;
doc[1] = 2;
```

However, this second syntax is a little slower because it requires walking the list of members. Use this syntax to *replace* elements and `add()` to append elements to the array.

Now that we can create an array, let's rewind a little because that's not the JSON array we want: instead of two integers, we need two nested objects.

4.3.3 Adding nested objects

To add the nested objects to the array, we call `JsonArray::add<JsonObject>()`. This function returns a reference to the newly created object, so we can use the subscript operator (`[]`) to add members.

Here is how to create our sample document:

```
JsonObject obj1 = doc.add<JsonObject>();
obj1["key"] = "a1";
obj1["value"] = analogRead(A1);

JsonObject obj2 = doc.add<JsonObject>();
obj2["key"] = "a2";
obj2["value"] = analogRead(A2);
```

Alternatively, we can create the same document like so:

```
doc[0]["key"] = "a1";
doc[0]["value"] = analogRead(A1);

doc[1]["key"] = "a2";
doc[1]["value"] = analogRead(A2);
```

Again, this syntax is slower because it needs to walk the list, so only use it for small documents.

4.3.4 Creating an empty array

We saw that the `JsonDocument` becomes an array as soon as we add elements, but this doesn't allow creating an empty array. If we want to create an empty array, we need to convert the `JsonDocument` explicitly with `JsonDocument::to<JsonArray>()`:

```
// Convert the JsonDocument to an array
JsonArray arr = doc.to<JsonArray>();
```

Now, the `JsonDocument` serializes to `[]`.

As we already saw, `JsonDocument::to<T>()` clears the `JsonDocument`, invalidating all previously acquired references.

4.3.5 Replacing and removing elements

As for objects, it's possible to replace elements in arrays using `JsonArray::operator[]`:

```
arr[0] = 666;
arr[1] = 667;
```

Finally, you can remove an element from the array with `JsonArray::remove()`:

```
arr.remove(0);
```

4.4 Writing to memory

We saw how to construct an array. Now, it's time to serialize it into a JSON document. There are several ways to do that. We'll start with a JSON document in memory.

We could use a `String`, but as you know, I'm not a big fan of this class, so instead, we'll use a plain old C string:

```
// Declare a buffer to hold the result
char output[128];
```

4.4.1 Minified JSON

To produce a JSON document from a `JsonDocument`, we simply need to call `serializeJson()`:

```
// Produce a minified JSON document
serializeJson(doc, output);
```

After this call, the string `output` contains:

```
[{"key":"a1","value":12},{"key":"a2","value":34}]
```

As you see, there are neither space nor line breaks; it's a "minified" JSON document.

4.4.2 Specifying (or not) the buffer size

If you're a C programmer, you may have been surprised I didn't provide the buffer size to `serializeJson()`. Indeed, there is an overload of `serializeJson()` that takes a `char*` and a size:

```
serializeJson(doc, output, sizeof(output));
```

However, that's not the overload we called in the previous snippet. Instead, we called a template method that infers the buffer size from its type (in this case, `char[128]`).

Of course, this shorter syntax only works because `output` is an array. If it were a `char*` or a variable-length array, we would have had to specify the size.

Variable-length array

A variable-length array, or VLA, is an array whose size is unknown at compile time. Here is an example:

```
void f(int n) {
  char buf[n];
  // ...
}
```

C99 and C11 allow VLAs, but not C++. However, some compilers support VLAs as an extension.

This feature is often criticized in C++ circles, but Arduino users seem to love it, so ArduinoJson supports VLAs in all functions that accept a string.

4.4.3 Prettified JSON

The minified version is what you use to store or transmit a JSON document because the size is optimal. However, it's not very easy to read. Humans prefer "prettified" JSON documents with spaces and line breaks.

To produce a prettified document, you must use `serializeJsonPretty()` instead of `serializeJson()`:

```
// Produce a prettified JSON document
serializeJsonPretty(doc, output);
```

Here is the content of `output`:

```
[
  {
    "key": "a1",
    "value": 12
  },
  {
    "key": "a2",
```

```
    "value": 34
  }
]
```

Of course, you need to make sure that the output buffer is big enough; otherwise, the JSON document will be truncated.

4.4.4 Measuring the length

ArduinoJson allows computing the length of the JSON document before producing it. This information is helpful for:

1. allocating an output buffer,
2. reserving the size on disk,
3. setting the `Content-Length` header.

There are two methods, depending on the type of document you want to produce:

```
// Compute the length of the minified JSON document
int len1 = measureJson(doc);

// Compute the length of the prettified JSON document
int len2 = measureJsonPretty(doc);
```

In both cases, the result doesn't count the null-terminator.

By the way, `serializeJson()` and `serializeJsonPretty()` return the number of bytes they wrote. The results are the same as `measureJson()` and `measureJsonPretty()`, except if the output buffer is too small.

Avoid prettified documents

With the example above, the sizes are `73` and `110`. In this case, the prettified version is only 50% bigger because the document is simple, but in most cases, the ratio is largely above 100%.

Remember, we're in an embedded environment: every byte counts, and so does every CPU cycle, so prefer minified documents.

4.4.5 Writing to a String

The functions `serializeJson()` and `serializeJsonPretty()` have overloads taking a `String`:

```
String output;
serializeJson(doc, output);
```

Of course, this also works with `std::string`.

4.4.6 Casting a JsonVariant to a String

You should remember from the chapter on deserialization that we must cast `JsonVariant` to the type we want to read. This feature also works for `String`, except the behavior is slightly different.

If the `JsonVariant` contains a string, the return value is a copy of the string. However, if the `JsonVariant` contains something else, the returned string is a serialization of the variant.

For example, we could rewrite the previous snippet like this:

```
// Cast the JsonDocument to a string
String output = doc.as<String>();
```

This trick works with `JsonDocument` and `JsonVariant` but not with `JsonArray` and `JsonObject` because they don't have an `as<T>()` function.

4.5 Writing to a stream

4.5.1 What's an output stream?

For now, every JSON document we produced remained in memory, but that's usually not what we want. In many situations, it's possible to send the JSON document directly to its destination (whether it's a file, a serial port, or a network connection) without any copy in RAM.

In the previous chapter, we saw what an "input stream" is, and we saw that Arduino represents this concept with the `Stream` class. Similarly, there are "output streams," which are sinks of bytes. We can write to an output stream, but we cannot read. In the Arduino land, an output stream is materialized by the `Print` class.

Here are examples of classes derived from `Print`:

Library	Class	Well known instances
Core	`HardwareSerial`	`Serial`, `Serial1`...
ESP	`BluetoothSerial`	`SerialBT`
	`File`	
	`WiFiClient`	
	`WiFiClientSecure`	
Ethernet	`EthernetClient`	
	`EthernetUDP`	
GSM	`GSMClient`	
LiquidCrystal	`LiquidCrystal`	
SD	`File`	
SoftwareSerial	`SoftwareSerial`	
WiFi	`WiFiClient`	
Wire	`TwoWire`	`Wire`

`std::ostream`

In the C++ Standard Library, an output stream is represented by the `std::ostream` class.

ArduinoJson supports both `Print` and `std::ostream`.

Performance issues

`serializeJson()` writes bytes one by one to the output stream, which can result in bad performances with unbuffered streams like `WiFiClient` or `File`.

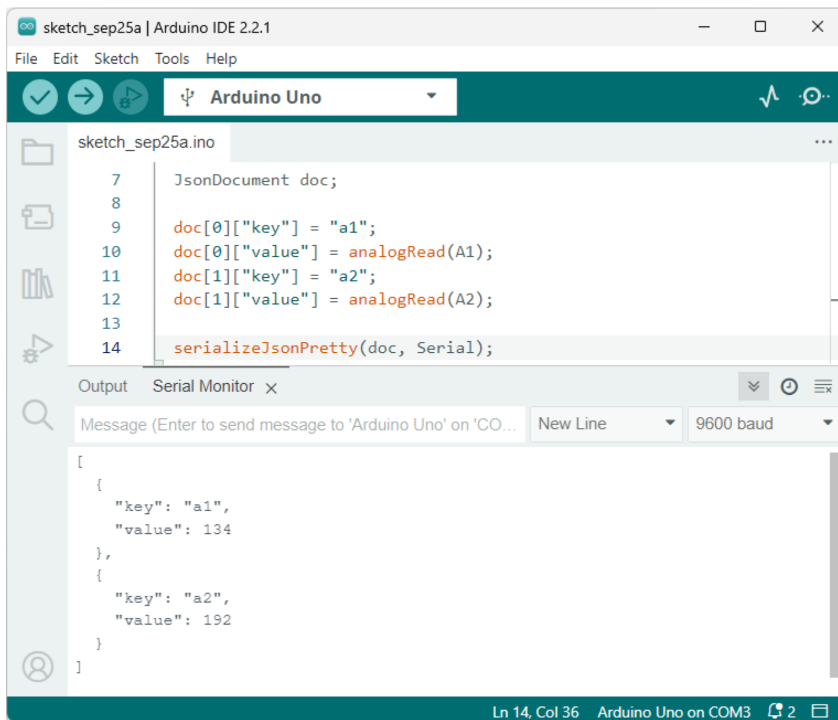
We'll see a simple workaround in the next chapter.

4.5.2 Writing to the serial port

The most famous implementation of `Print` is `HardwareSerial`, the class of `Serial`. To serialize a `JsonDocument` to the serial port of your Arduino, just pass `Serial` to `serializeJson()`:

```
// Print a minified JSON document to the serial port
serializeJson(doc, Serial);

// Same with a prettified document
serializeJsonPretty(doc, Serial);
```

You can see the result in the Arduino Serial Monitor, which is very handy for debugging.



If you want to send JSON documents between two boards, I recommend using `Serial1` for the communication link and keeping `Serial` for the debugging link. Of course, this requires that your board has several UARTs, which is not the case of our UNO R3.

Alternatively, you can use `Wire` for the communication link, but you must know that the Wire library limits the size of a message to 32 bytes (but there is a workaround for longer messages).

In theory, `SoftwareSerial` could also serve as the communication link, but I highly recommend against it because it's completely unreliable.

4.5.3 Writing to a file

Similarly, we can use a `File` instance as the target of `serializeJson()` and `serializeJsonPretty()`. Here is an example with the SD library:

```
// Open file for writing
File file = SD.open("adafruit.txt", FILE_WRITE);
```

```
// Write a prettified JSON document to the file
serializeJsonPretty(doc, file);
```

You can find the complete source code for this example in the `WriteSdCard` folder of the zip file provided with the book.

You can apply the same technique to write a file on SPIFFS or LittleFS, as we'll see in the case studies.

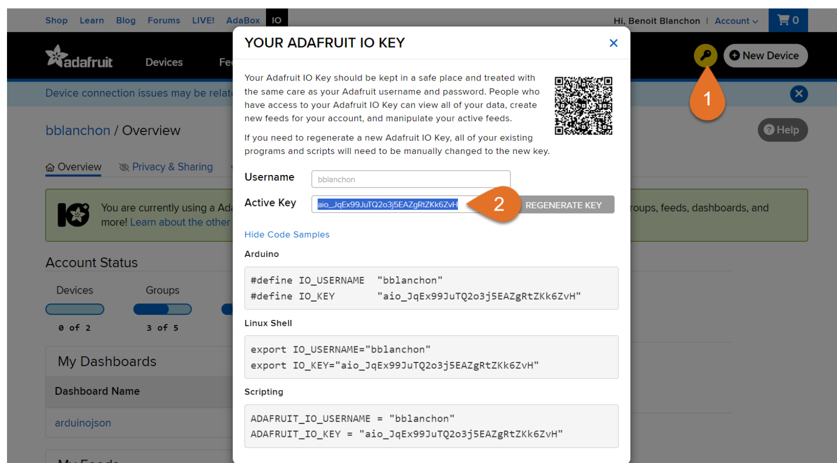
4.5.4 Writing to a TCP connection

We're now reaching our goal of sending our measurements to Adafruit IO.

As I said in the introduction, we'll suppose our program runs on an Arduino UNO with an Ethernet shield.

Preparing the Adafruit IO account

To run this program, you need an account on Adafruit IO (a free account is sufficient).



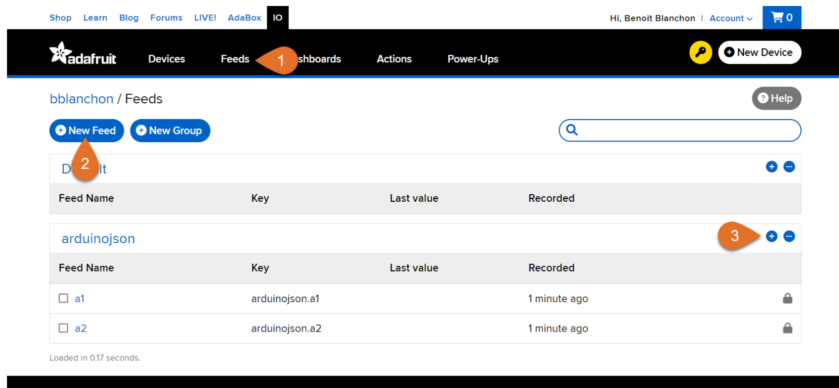
Then, you must copy your user name and “AIO key” to the source code.

```
#define IO_USERNAME "bblanchon"
#define IO_KEY "aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH"
```

We'll include the AIO key in an HTTP header, which will authenticate our program on Adafruit's server:

```
X-AIO-Key: aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH
```

Finally, you need to create a “group” named “arduinojson” in your Adafruit IO account. In this group, you must create two feeds: “a1” and “a2.”



The request

To send our measured samples to Adafruit IO, we have to send a `POST` request to `http://io.adafruit.com/api/v2/bblanchon/groups/arduinojson/data`, and include the following JSON document in the body:

```
{
  "location": {
    "lat": 48.748010,
    "lon": 2.293491
  },
  "feeds": [
    {
```

```
      "key": "a1",
      "value": 42
    },
    {
      "key": "a2",
      "value": 43
    }
  ]
}
```

As you see, it's a little more complex than our previous example because the array is not at the root of the document. Instead, the array is nested in an object under the key `"feeds"`.

Let's review the HTTP request before jumping to the code:

```
POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.0
Host: io.adafruit.com
Connection: close
Content-Length: 103
Content-Type: application/json
X-AIO-Key: aio_JqEx99JuTQ2o3j5EAZgRtZKk6ZvH

{"location":{"lat":48.748010,"lon":2.293491},"feeds":[{"key":"a1",...
```

The code

OK, time for action! We'll open a TCP connection to `io.adafruit.com` using an `EthernetClient` and send the request. As far as ArduinoJson is concerned, there are very few changes compared to the previous examples because we can pass the `EthernetClient` as the target of `serializeJson()`. We'll call `measureJson()` to set the value of the `Content-Length` header.

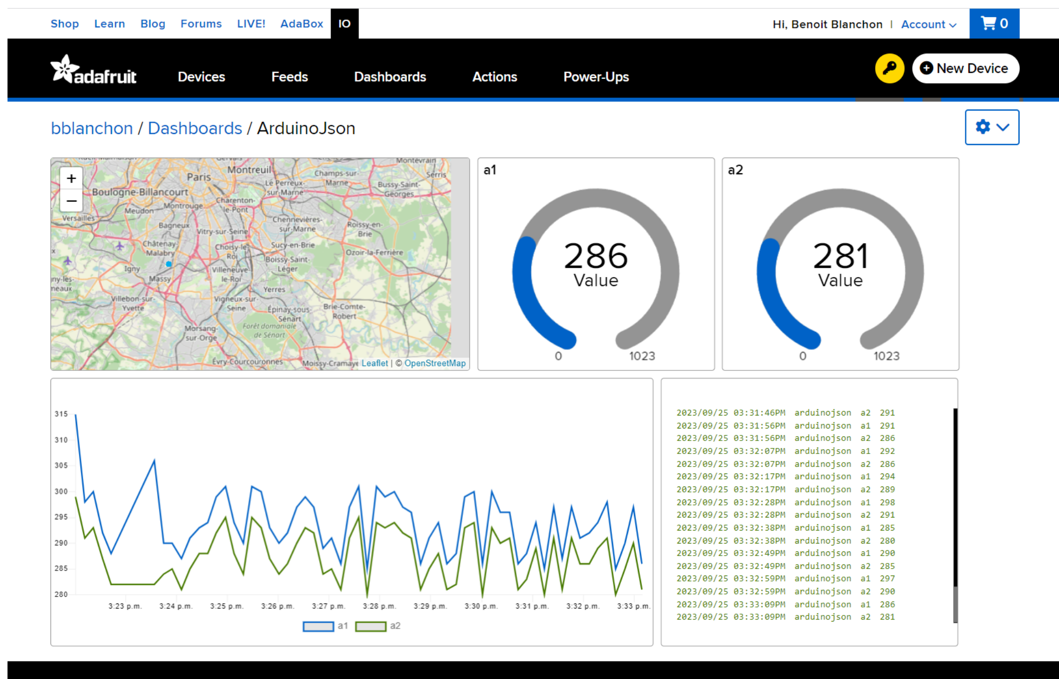
Here is the code:

```
// Create an empty document
JsonDocument doc;
```

```
// Add the "location" object
JsonObject location = doc["location"].to<JsonObject>();
location["lat"] = 48.748010;
location["lon"] = 2.293491;

// Add the "feeds" array
JsonArray feeds = doc["feeds"].to<JsonArray>();
JsonObject feed1 = feeds.add<JsonObject>();
feed1["key"] = "a1";
feed1["value"] = analogRead(A1);
JsonObject feed2 = feeds.add<JsonObject>();
feed2["key"] = "a2";
feed2["value"] = analogRead(A2);

// Connect to the HTTP server
EthernetClient client;
client.connect("io.adafruit.com", 80);

// Send "POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.0"
client.println("POST /api/v2/" IO_USERNAME
               "/groups/arduinojson/data HTTP/1.0");

// Send the HTTP headers
client.println("Host: io.adafruit.com");
client.println("Connection: close");
client.print("Content-Length: ");
client.println(measureJson(doc));
client.println("Content-Type: application/json");
client.println("X-AIO-Key: " IO_KEY);

// Terminate headers with a blank line
client.println();

// Send JSON document in body
serializeJson(doc, client);
```

You can find the complete source code of this example in the `AdafruitIo` folder of the zip file. This code includes the necessary error-checking code I removed from the book for clarity.

Below is a picture showing the results on the Adafruit IO dashboard.



4.6 Duplication of strings

Depending on the type, ArduinoJson stores strings either by pointer or copy. If the string is a `const char*`, it stores a pointer; otherwise, it makes a copy. This feature reduces memory consumption when you use string literals.

String type	Storage
`const char*`	pointer
`char*`	copy
`String`	copy
`const __FlashStringHelper*`	copy

ArduinoJson will store only one copy of each string, a feature called "string deduplication". For example, if you insert the string `"hello"` multiple times, the `JsonDocument` will only keep one copy.

4.6.1 An example

Compare this program:

```
// Create the array ["value1","value2"]
doc.add("value1");
doc.add("value2");
```

with the following:

```
// Create the array ["value1","value2"]
doc.add(String("value1"));
doc.add(String("value2"));
```

They both produce the same JSON document, but the second one consumes more memory because ArduinoJson copies the strings. For example, on an 8-bit microcontroller, the array consumes an additional 12 bytes. On a 32-bit microcontroller, it would take 30 extra bytes.

4.6.2 Keys and values

The duplication rules apply equally to keys and values. In practice, we mostly use string literals for keys, so they are rarely duplicated. String values, however, often originate from variables and entail string duplication.

Here is a typical example:

```
String identifier = getIdentifier();
doc["id"] = identifier;  // "id" is stored by pointer
                         // identifier is copied
```

Again, the duplication occurs for any type of string except `const char*`.

4.6.3 Copy only occurs when adding values

In the example above, ArduinoJson copied the `String` because it needed to add it to the `JsonDocument`. On the other hand, if you use a `String` to extract a value from a `JsonDocument`, it doesn't make a copy.

Here is an example:

```
// The following line produces a copy of "key"
doc[String("key")] = "value";

// The following line produces no copy
const char* value = doc[String("key")];
```

4.7 Inserting special values

Before finishing this chapter, let's see how we can insert special values in the JSON document.

4.7.1 Adding null

The first special value is `null`, which is a legal token in a JSON. There are several ways to add a `null` in a `JsonDocument`; here they are:

```
// Use a nullptr
arr.add(nullptr);

// Use a null char-pointer
arr.add((char*)0);

// Use a null JsonArray, JsonObject, or JsonVariant
arr.add(JsonVariant());
```

4.7.2 Adding pre-formatted JSON

The other special value is a JSON string that is already formatted and that ArduinoJson should not treat as a regular string.

You can do that by wrapping the string with a call to `serialized()`:

```
// adds "[1,2]"
arr.add("[1,2]");

// adds [1,2]
arr.add(serialized("[1,2]"));
```

The program above produces the following JSON document:

```
[
  "[1,2]",
  [1,2]
]
```

Use this feature when a part of the document cannot change; it will simplify your code and reduce the executable size. You can also use it to insert something the library doesn't allow.

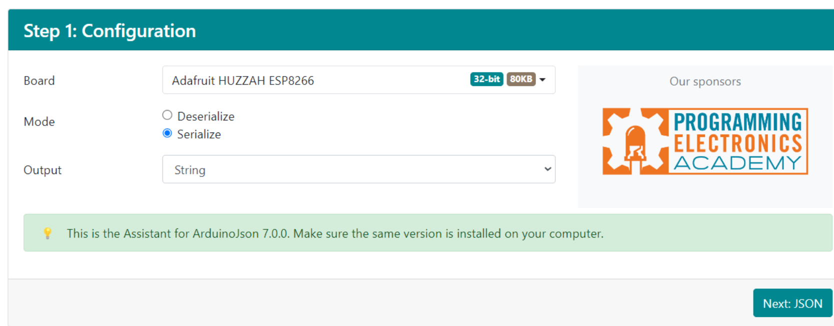
You can pass a Flash string or a `String` instance to `serialized()`. As usual, Flash strings must have the type `const __FlashStringHelper*` to be recognized as such.

Unlike regular ones, strings marked with `serialized()` are always stored by copy, even if they are `const char*`.

4.8 The ArduinoJson Assistant

In the previous chapter, we saw how the ArduinoJson Assistant could help us deserialize a JSON document. Now, we'll see how it can help us serialize a JSON document.

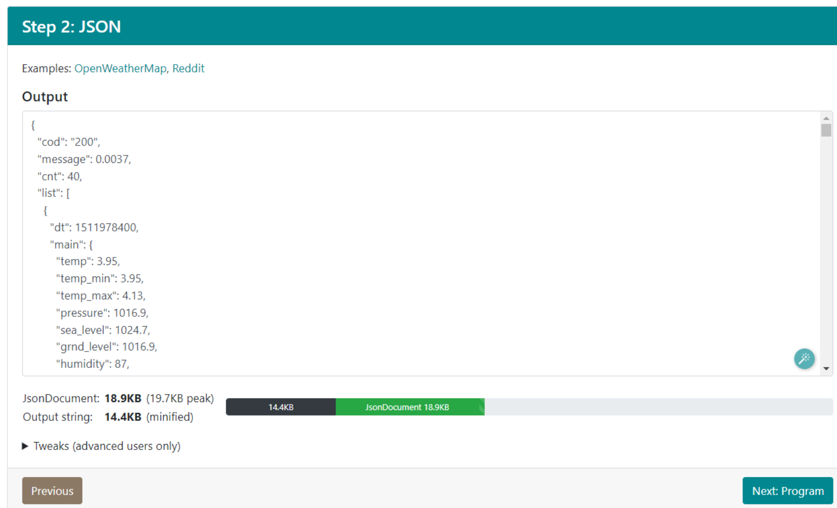
4.8.1 Step 1: Configuration



As we saw in the last chapter, the first step is to choose the board and the mode (serialization or deserialization), but instead of choosing the input type, we must choose the output type.

Again, these settings affect the memory consumption and the code generated in the last step.

4.8.2 Step 2: JSON



Step 2 changes slightly between serialization and deserialization. First, there is no filter option for serialization, and second, the advanced settings hidden in the "Tweaks" section are different. Again, we'll talk about them in the next chapter.

4.8.3 Step 3: Program

In the last step, the Assistant generates the code to serialize the JSON document you entered in the previous step.

You'll notice that contrary to deserialization, the Assistant doesn't offer any customization for the output. However, it writes the program according to the output type you chose in the first step.

4.9 Summary

In this chapter, we saw how to serialize a JSON document with ArduinoJson. Here are the key points to remember:

- Creating the document:
  - To add a member to an object, use the subscript operator (`obj[key] = value`).
    - Call `obj[key].to<JsonArray>()` to add a nested array.
    - Call `obj[key].to<JsonObject>()` to add a nested object.
  - The first time you add a member to a `JsonDocument`, it automatically becomes an object.
  - To append an element to an array, call `add()`.
    - Call `arr.add<JsonArray>()` to append a nested array.
    - Call `arr.add<JsonObject>()` to append a nested object.
  - The first time you append an element to a `JsonDocument`, it automatically becomes an array.
  - You can explicitly convert a `JsonDocument` with `JsonDocument::to<T>()`.
  - `JsonDocument::to<T>()` clears the `JsonDocument`, which invalidates all previously acquired references.
  - `JsonDocument::to<T>()` returns a reference to the root array or object.
  - When you insert a string in a `JsonDocument`, it makes a copy, except if it's a `const char*`.
- Serializing the document:
  - To serialize a `JsonDocument`, call `serializeJson()` or `serializeJsonPretty()`.
  - To compute the length of the JSON document, call `measureJson()` or `measureJsonPretty()`.
  - `serializeJson()` appends to `String`, but it overrides the content of a `char*`.
  - You can pass an instance of `Print` (like `Serial`, `EthernetClient`, `WiFiClient`, or `File`) to `serializeJson()` to avoid a copy in the RAM.

- The ArduinoJson Assistant is an online tool that:
  - Computes the memory consumption of your program.
  - Checks that the memory consumption is within the limits of your board.
  - Generates the code to serialize a JSON document.

In the next chapter, we'll see advanced techniques like filtering and logging.

Continue reading...

That was a free chapter from "Mastering ArduinoJson"; the book contains seven chapters like this one. Here is what readers say:

> This book is 100% worth it. Between solving my immediate problem in minutes, Chapter 2, and the various other issues this book made solving easy, **it is totally worth it**. I build software but I work in managed languages and for someone just getting started in C++and embedded programming this book has been indispensable. — Nathan Burnett

> I think the missing C++course and the troubleshooting chapter **are worth the money by itself**. Very useful for C programming dinosaurs like myself. — Doug Petican

> The short C++section was a great refresher. The practical use of ArduinoJson in small embedded processors was just what I needed for my home automation work. **Certainly worth having!** Thank you for both the book and the library. — Douglas S. Basberg

For a really reasonable price, not only you'll learn new skills, but you'll also be one of the few people that **contribute to sustainable open-source software**. Yes, giving money for free software is a political act!

The e-book comes in PDF and epub formats. If you purchase the e-book, **you get access to newer versions for free**. A carefully edited paperback edition is also available.

Ready to jump in?
Go to `arduinojson.org/book` and use the coupon code `THIRTY` to get a **30% discount**.

Thank you for your support!
Benoît