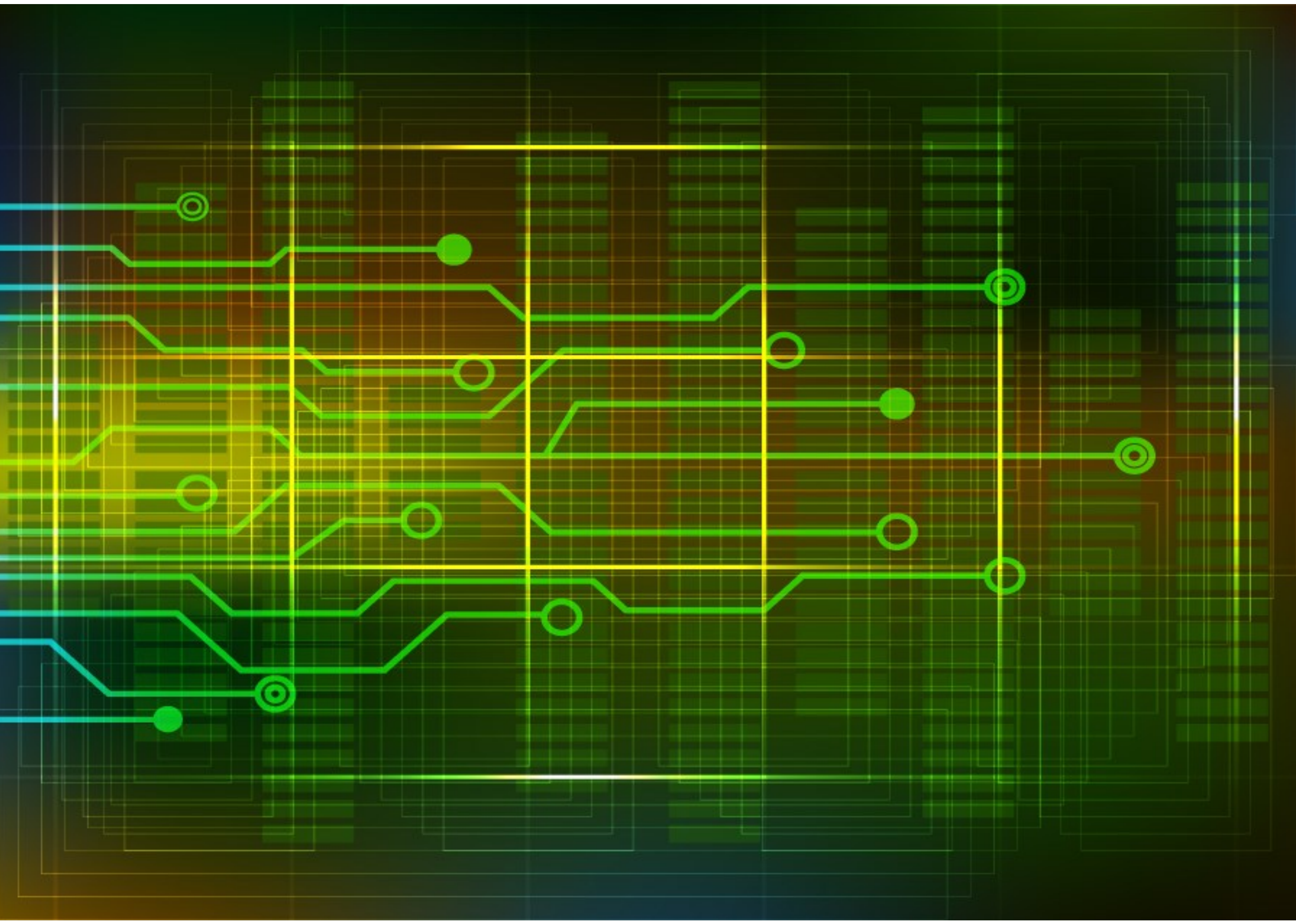
BENOIT BLANCHON
CREATOR OF ARDUINOJSON



Mastering ArduinoJson

Efficient JSON serialization for embedded C++

ArduinoJson

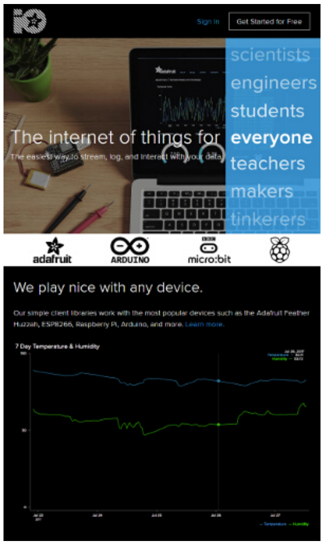
Contents

Chapter 4
Serialize 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 of the story; we'll now see how to write a JSON document with ArduinoJson.



The internet of things for everyone

The easiest way to stream, log, and interact with your data.

We play nice with any device.

Our simple client libraries work with the most popular devices such as the Adafruit Feather Huzzah, ESP8266, Raspberry Pi, Arduino, and more. Learn more.

The previous chapter revolved around weather forecast for the city of New York. We'll use a very different example for this chapter: pushing data to Adafruit IO.

Adafruit IO is a cloud storage for IoT data. They have a free plan with the following restrictions:

- 30 data points per minutes
- 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.

4.2 Create an object

4.2.1 The example

Here is the JSON object we want to create:

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

It's a flat object, meaning it has no nested object or array, and it contains the following piece of information:

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

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

4.2.2 Allocate the JsonBuffer

As for the deserialization, we start by creating a `JsonBuffer` to hold the in-memory representation of the object. The previous chapter introduces the `JsonBuffer`; please go back and read "Introducing `JsonBuffer`" if needed, as we won't repeat here.

We need to compute the capacity of the `JsonBuffer`. As the JSON document is simple, we can do the computation manually. We just have one object with no nested values, so the capacity is `JSON_OBJECT_SIZE(3)`. Remember that you can use ArduinoJson Assistant when the JSON document is more elaborate.

It's a reasonably small `JsonBuffer`, which fits in the stack on any microcontroller, so we can use a `StaticJsonBuffer`.

Here is the code:

```
const int capacity = JSON_OBJECT_SIZE(3);
StaticJsonBuffer<capacity> jb;
```

4.2.3 Create the object

It's not possible to directly instantiate a `JsonObject`, we'll see why in the next chapter. Instead, we need to ask the `JsonBuffer` to create one for us:

```
// Create a JsonObject
JsonObject& obj = jb.createObject();
```

This statement creates an empty object; its memory usage is currently `JSON_OBJECT_SIZE(0)`

As the object is located inside the `JsonBuffer`, we receive a reference to it. We saw the same thing with `parseObject()` in the previous chapter.

The factory design pattern

`JsonObject` cannot be constructed directly, but `JsonBuffer` provides a method to create an object.

It is an implementation of the "factory" design pattern: `JsonBuffer` is a factory of `JsonObject`.

4.2.4 Add the values

Adding values to a `JsonObject` is very similar to reading them.

There are several syntaxes, but the simplest is to use the subscript operator (`[]`):

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

The object's memory usage is now `JSON_OBJECT_SIZE(3)`, meaning that the `JsonBuffer` is full. When the `JsonBuffer` is full, you cannot add more values to the object, so don't forget to increase the capacity if you need.

4.2.5 Second syntax

With the syntax presented above, it's not possible to tell if the insertion succeeded or failed. Let's see another syntax:

```
obj.set("value", 42);
obj.set("lat", 48.748010);
obj.set("lon", 2.293491);
```

The executable generated by the compiler is the same as with the previous syntax, except that you can check if the insertion succeeded. You can check the result of `JsonObject::set()`, which will be `true` for success or `false` for failure. Again, insertion fails if the `JsonBuffer` is full.

Personally, I never check if insertion succeeds 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.

4.2.6 Third syntax

The syntax we just saw used `JsonObject::set()`. We can combine both syntax by using `JsonVariant::set()`:

```
obj["value"].set(42);
obj["lat"].set(48.748010);
obj["lon"].set(2.293491);
```

Again, the compiled executable is the same, and you can check the return value too.

4.2.7 Replace values

It's possible to replace a value in the object, for example:

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

It doesn't require a new allocation in the `JsonBuffer`, so if the first insertion succeeds, the second will succeed too.

4.2.8 Remove values

It's possible to erase values from an object by calling `JsonObject::remove(key)`. However, for reasons that will become clear in the next chapter, this function doesn't release the memory in the `JsonBuffer`.

The `remove()` function is a frequent cause of bugs because it creates a memory leak in the program. Indeed, if you add and remove values in a loop, the `JsonBuffer` grows, but memory is never released.

Code smell

In practice, this problem only happens in programs that use ArduinoJson to store the state of the application, which is not what ArduinoJson is for. Trying to optimize this use-case would inevitably impact the size and speed of ArduinoJson. Be careful not to fall into this common anti-pattern and make sure you read the case studies to see how ArduinoJson should be used.

4.3 Create an array

4.3.1 The example

Our next step will be to construct an array containing two objects:

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

4.3.2 Allocate the `JsonBuffer`

As usual, we start by computing the capacity of the `JsonBuffer`:

- There is one array with two elements: `JSON_ARRAY_SIZE(2)`
- There are two objects with two pairs: `2*JSON_OBJECT_SIZE(2)`

Here is the code:

```
const int capacity = JSON_ARRAY_SIZE(2) + 2*JSON_OBJECT_SIZE(2);
StaticJsonBuffer<capacity> jb;
```

4.3.3 Create the array

We create arrays the same way we create objects, by using the `JsonBuffer` as a factory:

```
JsonArray& arr = jb.createArray();
```

4.3.4 Add values

To add the nested objects to the array, we could create the two objects and add them to the array like that:

```
JsonObject& obj1 = jb.createObject();
obj1["key"] = "a1";
obj1["value"] = analogRead(A1);

JsonObject& obj2 = jb.createObject();
obj2["key"] = "a2";
obj2["value"] = analogRead(A2);

arr.add(obj1);
arr.add(obj2);
```

`JsonArray::add()` adds a new value at the end of the array. In this case, we added two `JsonObject`s, but you can use any value that a `JsonVariant` supports: `int`, `float`...

We just saw one way to create an array of object, but we can do better. Instead, we can call `JsonArray::createNestedObject()`, which creates the nested object and adds it to the end of the array. You can see on the snippet below that it leads to a simpler code:

```
JsonObject& obj1 = arr.createNestedObject();
obj1["key"] = "a1";
obj1["value"] = analogRead(A1);

JsonObject& obj2 = arr.createNestedObject();
obj2["key"] = "a2";
obj2["value"] = analogRead(A2);
```

Thanks to this technique, the program is shorter, faster, and more readable.

4.3.5 Replace values

As for objects, it's possible to replace values in arrays using either `JsonArray::operator[]` or `JsonArray::set()`:

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

Replacing the value doesn't require a new allocation in the `JsonBuffer`. However, if there was memory hold by the previous value, for example, a `JsonObject`, this memory is not released. Doing so would require counting references to the nested `JsonObject`, which ArduinoJson does not.

4.3.6 Remove values

As for objects, you can delete a slot of the array, by using `JsonArray::remove()`:

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

As described in the previous section, `remove()` doesn't release the memory from the `JsonBuffer`. You should never call this function in a loop.

4.3.7 Add `null`

To conclude this section, let's see how we can insert special values in the JSON document.

The first special value is `null`, which is a legal token in a JSON document. In ArduinoJson, it is a string whose address is zero:

```
// adds "null"
arr.add("null");

// add null
arr.add((char*)0);
```

The program above produces the following JSON document:

```
[
  "null",
  null
]
```

4.3.8 Add pre-formatted JSON

The other special value is a JSON element that is already formatted and that Arduino-Json should not treat as a string.

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

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

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

The program above produces the following JSON document:

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

4.4 Serialize to memory

We saw how to construct an array, and 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 may now start to see, I don't like using dynamic memory allocation. Instead, we'd use a good old `char[]`:

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

4.4.1 Minified JSON

Suppose we're still using our previous example for `JsonArray`, if we want to produce a JSON document out of it, we just need to call `JsonArray::printTo()`:

```
// Produce a minified JSON document
arr.printTo(output);
```

Now 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 Specify (or not) the size of the output buffer

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

```
arr.printTo(output, sizeof(output));
```

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

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

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 just need to use `prettyPrintTo()` instead of `printTo()`:

```
// Produce a prettified JSON document
arr.prettyPrintTo(output);
```

Here is the 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 incomplete.

4.4.4 Compute the length

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

1. allocating an output buffer,
2. reserving the size on disk, or
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 = arr.measureLength();

// Compute the length of the prettified JSON document
int len2 = arr.measurePrettyLength();
```

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

By the way, `printTo()` and `prettyPrintTo()` return the number of bytes written. Their return values are the same as `measureLength()` and `measurePrettyLength()`, except if the output buffer was too small.

Avoid prettified documents

The sizes in the example above are `73` and `110`. In this case, the prettified version is only 50% bigger because the document is simple. But, in most case, the ratio is largely above 100%. Remember, we're in an embedded environment: every byte counts and so does every CPU cycle. Always prefer a minified version.

4.4.5 Serialize to a `String`

The functions `printTo()` and `prettyPrintTo()` have overloads taking a `String`:

```
String output = "JSON = ";

arr.printTo(output);
```

The behavior is slightly different as the JSON document is appended to end the `String`. The snippet above sets the content of the `output` variable to:

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

4.4.6 Cast 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.

It is also possible to cast a `JsonVariant` to a `String`. If the `JsonVariant` contains a string, the return value is a copy of the string. However, if the `JsonVariant` contains something else, the return value is a serialization of the value.

We could rewrite the previous example like this:

```
// Wrap the JsonArray in a JsonVariant
JsonVariant v = arr;

// Cast the JsonVariant to a string
String output = "toto" + v.as<String>();
```

Unfortunately, this trick is only available on `JsonVariant`; you cannot do the same with `JsonArray` and `JsonObject` (unless you put them in a `JsonVariant`). Furthermore, this technique only produces a minified document.

4.5 Serialize to stream

4.5.1 What's an output stream?

In the previous section, we saw how to serialize an array or an object. 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.

We saw in the previous chapter what an input stream is, and that Arduino represents this concept with the class `Stream`.

Similar to input streams, we also have "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 class `Print`.

Here are examples of classes derived from `Print`:

Library	Class	Well known instances
Core	`HardwareSerial`	`Serial`, `Serial1`...
ESP8266 FS	`File`	
Ethernet	`EthernetClient`	
Ethernet	`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 class `std::ostream`. ArduinoJson supports both `Print` and `std::ostream`.

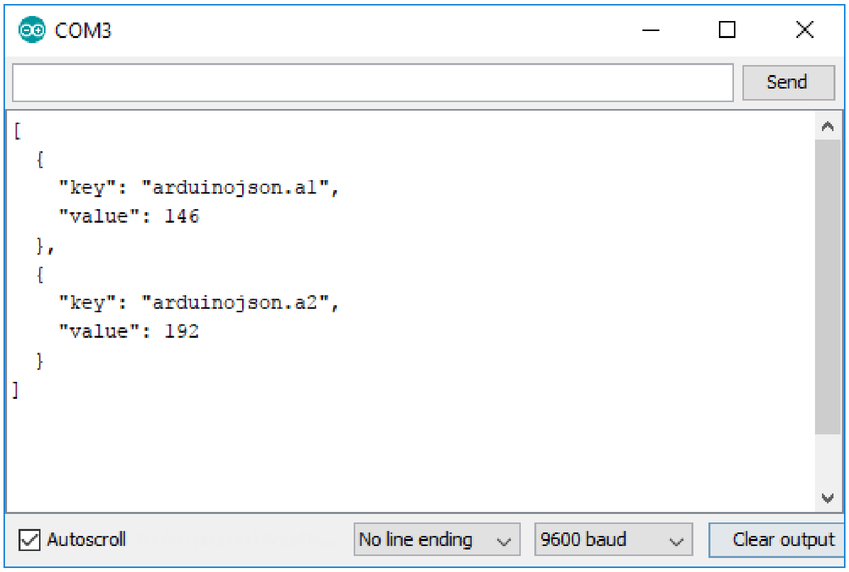
4.5.2 Serialize to `Serial`

The most famous implementation of `Print` is `HardwareSerial`, which is the class of `Serial`. To serialize a `JsonArray` or a `JsonObject` to the serial port of your Arduino, just pass `Serial` to `printTo()`:

```
// Print a minified version to the serial port
arr.printTo(Serial);

// Same with a prettified version
arr.prettyPrintTo(Serial);
```

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

There are also other serial port implementations that you can use this way, for example `SoftwareSerial` and `TwoWire`.



```
[
  {
    "key": "arduinojson.a1",
    "value": 146
  },
  {
    "key": "arduinojson.a2",
    "value": 192
  }
]
```

4.5.3 Serialize to a file

Similarly, we can use a `File` instance as the target of `printTo()` and `prettyPrintTo()`. 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
arr.prettyPrintTo(file);
```

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

You can apply the same technique to write a file on an ESP8266, as we'll see in the case studies.

4.5.4 Serialize to an HTTP request

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

To do that, we need to send the following JSON document:

```
{
  "location": {
    "lat": 48.748010,
    "lon": 2.293491
  },
  "feeds": [
    {
      "key": "a1",
      "value": 42
    },
    {
      "key": "a2",
      "value": 43
    }
  ]
}
```

This document contains the values to add to our two feeds `a1` and `a2`.

We will send this document in the body of the following HTTP request:

```
POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.1
Host: io.adafruit.com
Connection: close
Content-Length: 103
Content-Type: application/json
X-AIO-Key: baf4f21a32f6438eb82f83c3eed3f3b3

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

Here is the program:

```
// Allocate JsonBuffer
const int capacity = JSON_ARRAY_SIZE(2) + 4 * JSON_OBJECT_SIZE(2);
StaticJsonBuffer<capacity> jb;

// Create JsonObject
JsonObject &root = jb.createObject();

// Add location
JsonObject &location = root.createNestedObject("location");
location["lat"] = 48.748010;
location["lon"] = 2.293491;

// Add feeds array
JsonArray &feeds = root.createNestedArray("feeds");
JsonObject &feed1 = feeds.createNestedObject();
feed1["key"] = "a1";
feed1["value"] = analogRead(A1);
JsonObject &feed2 = feeds.createNestedObject();
feed2["key"] = "a2";
feed2["value"] = analogRead(A2);

// Connect to the HTTP server
EthernetClient client;
client.setTimeout(10000);
client.connect("io.adafruit.com", 80);

// Send "POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.1"
```

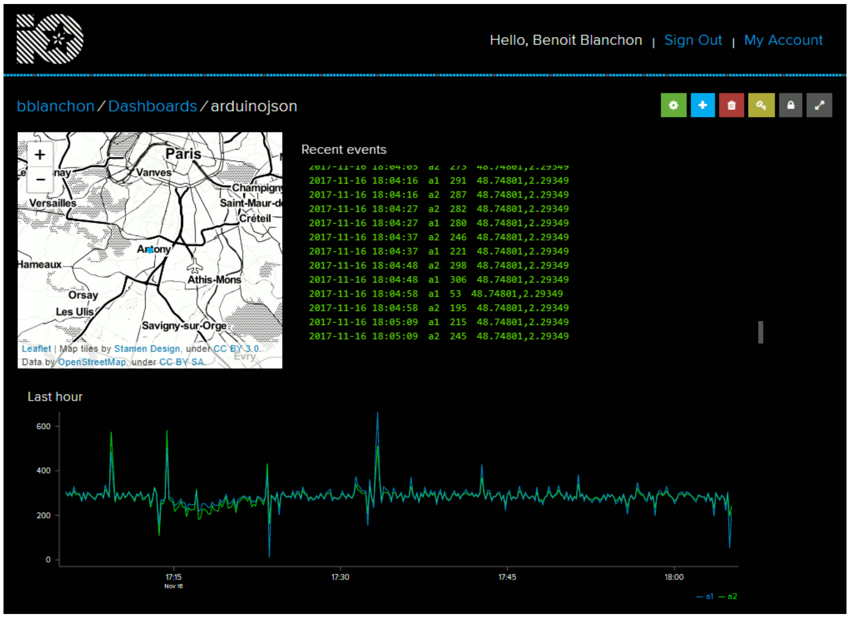
```
client.print("POST /api/v2/");
client.print(username);
client.print("/groups/");
client.print(groupname);
client.println("/data HTTP/1.1");

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

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

// Send JSON document in body
root.printTo(client);
```

You can find the complete source code of this example in the folder `AdafruitIo` in the zip file.

If you want to reproduce this example, you need to follow these steps:

1. Create an account on Adafruit IO (a free tier is sufficient).
2. Create a feed group named `arduinojson`.
3. In that group, create two feeds `a1` and `a2`.
4. Set the constant `username` to your user name.
5. Set the constant `apiKey` to your AIO key.

Below is a picture of a dashboard showing the data from this example.


Hello, Benoit Blanchon | Sign Out | My Account
bblanchon / Dashboards / arduinojson
Paris
Vanves
Versailles
Champigny
Saint-Maur-d
Créteil
Antony
Hameaux
Athis-Mons
Orsay
Les Ulis
Savigny-sur-Orge
Leaflet | Map tiles by Stamen Design, under CC BY 3.0. Data by OpenStreetMap, under CC BY SA.
Last hour
600
400
200
0
17:15 Nov 16
17:30
17:45
18:00
— a1 — a2

Recent events

```
2017-11-16 18:04:05  a2  275  48.74801,2.29349
2017-11-16 18:04:16  a1  291  48.74801,2.29349
2017-11-16 18:04:16  a2  287  48.74801,2.29349
2017-11-16 18:04:27  a2  282  48.74801,2.29349
2017-11-16 18:04:27  a1  280  48.74801,2.29349
2017-11-16 18:04:37  a2  246  48.74801,2.29349
2017-11-16 18:04:37  a1  221  48.74801,2.29349
2017-11-16 18:04:48  a2  298  48.74801,2.29349
2017-11-16 18:04:48  a1  306  48.74801,2.29349
2017-11-16 18:04:58  a1  53  48.74801,2.29349
2017-11-16 18:04:58  a2  195  48.74801,2.29349
2017-11-16 18:05:09  a1  215  48.74801,2.29349
2017-11-16 18:05:09  a2  245  48.74801,2.29349
```

4.6 Duplication of strings

When you add a string value to a `JsonArray` or a `JsonObject`, ArduinoJson either stores a pointer or a copy of the string depending on its type. If the string is a `const char*`, it stores a pointer; otherwise, it makes a copy.

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

As usual, the copy lives in the `JsonBuffer`, so you may need to increase its capacity depending on the type of string you use.

The table above reflects the new rules that are in place since ArduinoJson 5.13; on older versions `char*` were stored with a pointer, which caused surprising effects.

4.6.1 An example

Compare this program:

```
// Create the array ["value1","value2"]
JsonArray& arr = jb.createArray();
arr.add("value1");
arr.add("value2");

// Print the memory usage
Serial.println(jb.size());
```

with the following:

```
// Create the array ["value1","value2"]
JsonArray& arr = jb.createArray();
arr.add(String("value1"));
arr.add(String("value2"));
```

```
// Print the memory usage
Serial.println(jb.size());
```

They both produce the same JSON document, but the second one require much more memory because ArduinoJson has to make copies. If you run these programs on an ATmega328, you'll see `20` for the first one and `32` for the second.

4.6.2 Copy only occurs when adding values

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

Here is an example:

```
JsonObject& obj = jb.createObject();

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

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

4.6.3 Why copying Flash strings?

I understand that it is disappointing that ArduinoJson copies Flash strings into the `JsonBuffer`. Unfortunately, there are several situations where it needs to have the strings in RAM.

For example, if the user calls `JsonVariant::as<char*>()`, a pointer to the copy is returned:

```
// The value is originally in Flash memory
obj["hello"] = F("world");

// But the returned value is in RAM (in the JsonBuffer)
```

```
const char* world = obj["hello"];
```

It is required for `JsonPair` too. If the string is a key in an object and the user iterates through the object, the `JsonPair` contains a pointer to the copy:

```
// The key is originally in Flash memory
obj[F("hello")] = "world";

for(JsonPair& kvp : obj) {
    // But the key is actually stored in RAM (in the JsonBuffer)
    const char* key = kvp.key;
}
```

However, retrieving a value using a Flash string as a key doesn't cause a copy:

```
// The Flash string is not copied in this case
const char* world = obj[F("hello")];
```

Avoid Flash string with ArduinoJson

Storing strings in Flash is a great way to reduce RAM usage, but remember that they are copied into the `JsonBuffer`. If you wrap all your strings with `F()`, you'll need a much bigger `JsonBuffer`. Moreover, the program will waste a lot of time copying the string; it will be much slower than with conventional strings.

4.6.4 `RawJson()`

We saw earlier in this chapter that the `RawJson()` function marks strings as JSON segments that should not be treated as string values.

Since ArduinoJson 5.13, `RawJson()` supports all the string types (`char*`, `const char*`, `String` and `const __FlashStringHelper*`) and obeys to the rules stated in the table above.

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 three formats: PDF, epub and mobi. 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