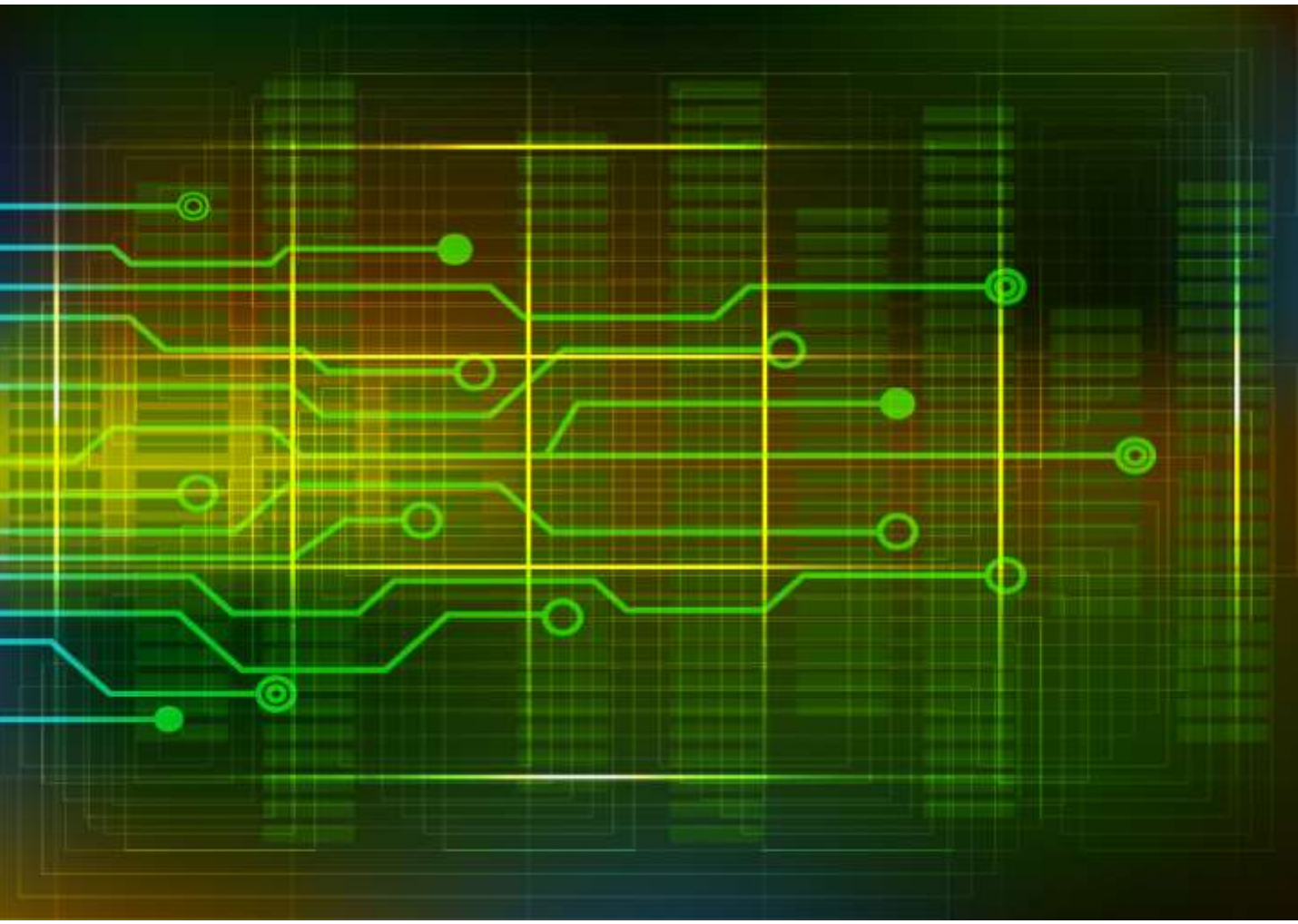
BENOIT BLANCHON
CREATOR OF ARDUINOJSON



Mastering ArduinoJson 6

Efficient JSON serialization for embedded C++

ArduinoJson

THIRD EDITION

Contents

Chapter 3

Deserialize with ArduinoJson

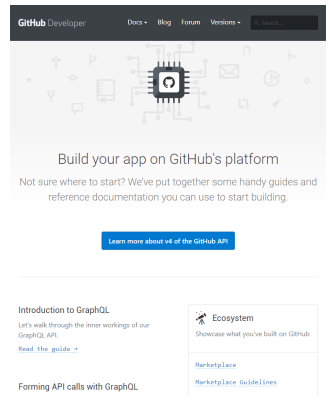
It is not the language that makes programs appear simple. It is the programmer that makes the language appear simple!

– Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

3.1 The example of this chapter

Now that you're familiar with JSON and C++, we're going to learn how to use ArduinoJson. This chapter explains everything there is to know about deserialization. As we've seen, deserialization is the process of converting a sequence of bytes into a memory representation. In our case, it means converting a JSON document to a hierarchy of C++ structures and arrays.

In this chapter, we'll use a JSON response from GitHub's API as an example. As you already know, GitHub is a hosting service for source code; what you may not know, however, is that GitHub provides a very powerful API that allows you to interact with the platform.



We could do many things with GitHub's API, but in this chapter, we'll only focus on a small part. We'll get your ten most popular repositories and display their names, numbers of stars, and numbers of opened issues.

There are several versions of GitHub's API; we'll use the latest one: the GraphQL API (or v4). We'll use this one because it allows us to get all the information we need with only one query. It also returns much smaller responses compared to v3, which is appreciable for embedded software.

If you want to run the example, you'll need a user account on GitHub and a personal access token. Don't worry; we'll see that later.

Because GitHub only allows secure connections, we need a microcontroller that supports HTTPS. We'll use the ESP8266 with the ESP8266HTTPClient as an example. If you want to use ArduinoJson with `EthernetClient`, `WiFiClient`, or `WiFiClientSecure`, check out the case studies in the last chapter.

Now that you know where we are going, we'll back up a few steps and start with a basic example. Then, we'll progressively learn new things so that we'll finally be able to interact with GitHub by the end of the chapter.

3.2 Deserializing an object

We'll begin this tutorial with the simplest situation: a JSON document in memory. More precisely, our JSON document resides in the stack in a writable location. This fact is going to matter, as we will see later.

3.2.1 The JSON document

Our example is the repository information for ArduinoJson:

```
{
  "name": "ArduinoJson",
  "stargazers": {
    "totalCount": 5246
  },
  "issues": {
    "totalCount": 15
  }
}
```

As you see, it's a JSON object that contains two nested objects. It includes the name of the repository, the number of stars, and the number of open issues.

3.2.2 Placing the JSON document in memory

In our C++ program, this JSON document translates to:

```
char input[] = "{\"name\":\"ArduinoJson\",\"stargazers\":{"
               "\"totalCount\":5246},\"issues\":{\"totalCount\":15}}"
```

In the previous chapter, we saw that this code creates a duplication of the string in the stack. We know it's a code smell in production code, but it's a good example for learning. This unusual construction creates a *writable* (i.e., not read-only) input string, which is essential for your first contact with ArduinoJson.

3.2.3 Introducing JsonDocument

As we saw in the introduction, one of the unique features of ArduinoJson is its fixed memory allocation strategy.

Here is how it works:

1. First, you create a `JsonDocument` to reserve a specified amount of memory.
2. Then, you deserialize the JSON document.
3. Finally, you destroy the `JsonDocument`, which releases the reserved memory.

The memory of the `JsonDocument` can be either in the stack or in the heap. The location depends on the derived class you choose. If you use a `StaticJsonDocument`, it will be in the stack; if you use a `DynamicJsonDocument`, it will be in the heap.

A `JsonDocument` is responsible for reserving and releasing the memory used by ArduinoJson. It is an instance of the RAII idiom that we saw in the previous chapter.

`StaticJsonDocument` in the heap

I often say that a `StaticJsonDocument` resides in the stack, but it's possible to have it in the heap, for example, if a `StaticJsonDocument` is a member of an object in the heap.

It's also possible to allocate a `StaticJsonDocument` with `new`, but I strongly advise against it because you would lose the RAII feature.

3.2.4 How to specify the capacity?

When you create a `JsonDocument`, you must specify its capacity in bytes.

In the case of `DynamicJsonDocument`, you set the capacity via a constructor argument:

```
DynamicJsonDocument doc(capacity);
```

Since it's a constructor parameter, you can use a regular variable whose value can change at run-time.

In the case of a `StaticJsonDocument`, you set the capacity via a template parameter:

```
StaticJsonDocument<capacity> doc;
```

As it's a template parameter, you cannot use a variable. Instead, you must use a constant expression, which means that the value must be computed at compile-time. As we said in the previous chapter, the compiler manages the stack, so it needs to know the size of each variable when it compiles the program; that's why we must use a constant expression here.

3.2.5 How to determine the capacity?

Now comes a tricky question for every new user of ArduinoJson: what should be the capacity of my `JsonDocument`?

To answer this question, you need to know what ArduinoJson stores in the `JsonDocument`. ArduinoJson needs to store a data structure that mirrors the hierarchy of objects in the JSON document. In other words, the `JsonDocument` contains objects which relate to one another the same way they do in the JSON document.

Therefore, the capacity of the `JsonDocument` highly depends on the complexity of the JSON document. If it's just one object with few members, like our example, a few dozens of bytes are enough. If it's a massive JSON document, like OpenWeatherMap's response, up to a hundred kilobytes are needed.

ArduinoJson provides macros for computing precisely the capacity of the `JsonDocument`. The macro to compute the size of an object is `JSON_OBJECT_SIZE()`. It takes one argument: the number of members in the object.

Here is how to compute the capacity for our sample document:

```
// Enough space for:
//  + 1 object with 3 members
//  + 2 objects with 1 member
const int capacity = JSON_OBJECT_SIZE(3) + 2 * JSON_OBJECT_SIZE(1);
```

On an ESP8266, a 32-bit processor, this expression evaluates to `80` bytes. The result would be significantly smaller on an 8-bit processor; for example, it would be `40` bytes on an ATmega328.

A read-only input requires a higher capacity

In this part of the tutorial, we consider the case of a writeable input because it simplifies the computation of the capacity. However, if the input is read-only (for example, a `const char*` instead of `char[]`), you must increase the capacity.

We'll talk about that later, in the section "Reading from read-only memory."

3.2.6 `StaticJsonDocument` or DynamicJsonDocument?

Since our `JsonDocument` is small, we can keep it in the stack. Using the stack, we reduce the executable size and improve the performance because we avoid the overhead due to the management of the heap.

Here is our program so far:

```
const int capacity = JSON_OBJECT_SIZE(3) + 2*JSON_OBJECT_SIZE(1);
StaticJsonDocument<capacity> doc;
```

Of course, if the `JsonDocument` were bigger, it would make sense to move it to the heap. We'll do that later.

Don't forget `const`!

If you forget to write `const`, the compiler produces the following error:

```
error: the value of 'capacity' is not usable in a constant
↪  expression
```

Indeed, a template parameter is evaluated at compile-time, so it must be a constant expression. By definition, a constant expression is computed at compile-time, as opposed to a variable, which is computed at run-time.

3.2.7 Deserializing the JSON document

Now that the `JsonDocument` is ready, we can parse the input with `deserializeJson()`:

```
DeserializationError err = deserializeJson(doc, input);
```

`deserializeJson()` returns a `DeserializationError` that tells whether the operation was successful. It can have one of the following values:

- `DeserializationError::Ok`: the deserialization was successful.
- `DeserializationError::EmptyInput`: the input was empty or contained only spaces.
- `DeserializationError::IncompleteInput`: the input was valid but ended prematurely.
- `DeserializationError::InvalidInput`: the input was not a valid JSON document.
- `DeserializationError::NoMemory`: the `JsonDocument` was too small.
- `DeserializationError::TooDeep`: the input was valid, but it contained too many nesting levels; we'll talk about that later in the book.

I listed all the error codes above so that you can understand how the library works; however, I don't recommend using them directly in your code.

First, `DeserializationError` converts implicitly to `bool`, so you don't have to write `if (err != DeserializationError::Ok)`; you can simply write `if (err)`.

Second, `DeserializationError` has a `c_str()` member function that returns a string representation of the error. It also has an `f_str()` member that returns a Flash string, saving some space on Harvard architectures like ESP8266.

Thanks to these two features of `DeserializationError`, you can simply write:

```
if (err) {
  Serial.print(F("deserializeJson() failed with code "));
  Serial.println(err.f_str());
}
```

In the "Troubleshooting" chapter, we'll look at each error code and see what can cause the error.

3.3 Extracting values from an object

In the previous section, we created a `JsonDocument` and called `deserializeJson()`, so now, the `JsonDocument` contains a memory representation of the JSON input. Let's see how we can extract the values.

3.3.1 Extracting values

There are multiple ways to extract the values from a `JsonDocument`; let's start with the simplest:

```
const char* name   = doc["name"];
long        stars  = doc["stargazers"]["totalCount"];
int         issues = doc["issues"]["totalCount"];
```

This syntax leverages two C++ features:

1. Operator overloading: the subscript operator (`[]`) has been customized to mimic a JavaScript object.
2. Implicit casts: the result of the subscript operator is implicitly converted to the type of the variable.

3.3.2 Explicit casts

Not everyone likes implicit casts, mainly because they mess with overload resolution, template parameter type deduction, and the `auto` keyword. That's why ArduinoJson offers an alternative syntax with explicit type conversion.

The `auto` keyword

The `auto` keyword is a feature of C++11. In this context, it allows inferring the type of the variable from the type of the expression on the right. It is the equivalent of `var` in C# and Java.

Here is the same code adapted for this school of thought:

```
auto name   = doc["name"].as<const char*>();
auto stars  = doc["stargazers"]["totalCount"].as<long>();
auto issues = doc["issues"]["totalCount"].as<int>();
```

Implicit or explicit?

We saw two different syntaxes to do the same thing. They are all equivalent and lead to the same executable.

I prefer the implicit version because it allows using the "or" operator, as we'll see. I use the explicit version only to solve ambiguities.

3.3.3 When values are missing

We saw how to extract values from an object, but we didn't do error checking. Let's see what happens when a value is missing.

When you try to extract a value that is not present in the document, ArduinoJson returns a default value. This value depends on the requested type:

Requested type	Default value
`const char*`	`nullptr`
`float, double`	`0.0`
`int, long…`	`0`
`String`	`""`
`JsonArray`	a null object
`JsonObject`	a null object

The two last lines (`JsonArray` and `JsonObject`) happen when you extract a nested array or object, we'll see that in a later section.

No exceptions

ArduinoJson never throws exceptions. Exceptions are an excellent C++ feature, but they produce large executables, which is unacceptable for microcontrollers.

3.3.4 Changing the default value

Sometimes, the default value from the table above is not what you want. In this situation, you can use the operator `|` to change the default value. I call it the "or" operator because it provides a replacement when the value is missing or incompatible.

Here is an example:

```
// Get the port or use 80 if it's not specified
short tcpPort = config["port"] | 80;
```

This feature is handy to specify default configuration values, like in the snippet above, but it is even more helpful to prevent a null string from propagating.

Here is an example:

```
// Copy the hostname or use "arduinojson.org" if it's not specified
char hostname[32];
strlcpy(hostname, config["hostname"] | "arduinojson.org", 32);
```

`strlcpy()`, a function that copies a source string to a destination string, crashes if the source is null. Without the operator `|`, we would have to use the following code:

```
char hostname[32];
const char* configHostname = config["hostname"];
if (configHostname != nullptr)
  strlcpy(hostname, configHostname, 32);
else
  strcpy(hostname, "arduinojson.org");
```

We'll see a complete example that uses this syntax in the case studies.

3.4 Inspecting an unknown object

In the previous section, we extracted the values from an object that we knew in advance. Indeed, we knew that the JSON object had three members: a string named "name," a nested object named "stargazers," and another nested object named "issues." In this section, we'll see how to inspect an *unknown* object.

3.4.1 Getting a reference to the object

So far, we have a `JsonDocument` that contains a memory representation of the input. A `JsonDocument` is a very generic container: it can hold an object, an array, or any other value allowed by JSON. Because it's a generic container, `JsonDocument` only offers methods that apply unambiguously to objects, arrays, and any other supported type.

For example, we saw that we could call the subscript operator (`[]`), and the `JsonDocument` happily returned the associated value. However, the `JsonDocument` cannot enumerate the object's member because it doesn't know, at compile-time, whether it should behave as an object or an array.

To remove the ambiguity, we must get the object within the `JsonDocument`. We do that by calling the member function `as<JsonObject>()`, like so:

```
// Get a reference to the root object
JsonObject obj = doc.as<JsonObject>();
```

And now, we're ready to enumerate the members of the object!

`JsonObject` has reference semantics

`JsonObject` is not a copy of the object in the document; on the contrary, it's a reference to the object in the `JsonDocument`. When you modify the `JsonObject`, you also alter the `JsonDocument`.

In a sense, we can say that `JsonObject` is a smart pointer. It wraps a pointer with a class that is easy to use. Unlike the other smart pointers we talked about in the previous chapter, `JsonObject` doesn't release the memory for the object when it goes out of scope because that's the role of the `JsonDocument`.

3.4.2 Enumerating the keys

Now that we have a `JsonObject`, we can look at all the keys and their associated values. In ArduinoJson, a key-value pair is represented by the type `JsonPair`.

We can enumerate all pairs with a simple for loop:

```
// Loop through all the key-value pairs in obj
for (JsonPair p : obj) {
  p.key() // is a JsonString
  p.value() // is a JsonVariant
}
```

Notice these three points about this code:

1. I explicitly used a `JsonPair` to emphasize the type, but you can use `auto`.
2. The value associated with the key is a `JsonVariant`, a type that can represent any JSON type.
3. You can convert the `JsonString` to a `const char*` with `JsonString::c_str()`.

When C++11 is not available

The code above leverages a C++11 feature called "range-based for loop". If you cannot enable C++11 on your compiler, you must use the following syntax:

```
for (JsonObject::iterator it=obj.begin(); it!=obj.end(); ++it) {
  it->key() // is a JsonString
  it->value() // is a JsonVariant
}
```

3.4.3 Detecting the type of value

Like `JsonObject`, `JsonVariant` is a reference to a value stored in the `JsonDocument`. However, `JsonVariant` can refer to any JSON value: string, integer, array, object… A `JsonVariant` is returned when you call the subscript operator, like `obj["text"]` (we'll see that this statement is not entirely correct, but for now, we can say it's a `JsonVariant`).

To know the actual type of the value in a `JsonVariant`, you need to call `JsonVariant::is<T>()`, where `T` is the type you want to test.

For example, the following snippet checks if the value is a string:

```
// Is it a string?
if (p.value().is<const char*>()) {
  // Yes!
  // We can get the value via implicit cast:
  const char* s = p.value();
  // Or, via explicit method call:
  auto s = p.value().as<const char*>();
}
```

If you use this with our sample document, you'll see that only the member "name" contains a string. The two others are objects, as `is<JsonObject>()` would confirm.

3.4.4 Variant types and C++ types

There are a limited number of types that a variant can use: boolean, integer, float, string, array, and object. However, different C++ types can store the same JSON type; for example, a JSON integer could be a `short`, an `int`, or a `long` in the C++ code.

The following table shows all the C++ types you can use as a parameter for `JsonVariant::is<T>()` and `JsonVariant::as<T>()`.

Variant type	Matching C++ types
Boolean	`bool`
Integer	`int`, `long`, `short`, `char` (all `signed` and `unsigned`)
Float	`float`, `double`
String	`const char*`, `String`, `std::string`
Array	`JsonArray`
Object	`JsonObject`

More on `arduinojson.org`

The complete list of types that you can use as a parameter for `JsonVariant::is<T>()` can be found in the API Reference.

3.4.5 Testing if a key exists in an object

If you have an object and want to know whether a key is present or not, you can call `JsonObject::containsKey()`.

Here is an example:

```
// Is there a value named "text" in the object?
if (obj.containsKey("text")) {
  // Yes!
}
```

However, I don't recommend using this function because you can avoid it most of the time.

Here is an example where we can avoid `containsKey()`:

```
// Is there a value named "error" in the object?
if (obj.containsKey("error")) {
  // Get the text of the error
  const char* error = obj["error"];
  // ...
}
```

The code above is not horrible, but it can be simplified and optimized if we just remove the call to `containsKey()`:

```
// Get the text of the error
const char* error = obj["error"];

// Is there an error after all?
if (error != nullptr) {
  // ...
}
```

This code is faster and smaller because it only looks for the key "error" once, whereas the previous code did it twice.

3.5 Deserializing an array

3.5.1 The JSON document

We've seen how to parse a JSON object from GitHub's response; it's time to move up a notch by parsing an array of objects. Indeed, our goal is to display the top 10 of your repositories, so there will be up to 10 objects in the response. In this section, we'll suppose that there are only two repositories, but you and I know that it will be more in the end.

Here is the new sample JSON document:

```
[
  {
    "name": "ArduinoJson",
    "stargazers": {
      "totalCount": 5246
    },
    "issues": {
      "totalCount": 15
    }
  },
  {
    "name": "pdfium-binaries",
    "stargazers": {
      "totalCount": 249
    },
    "issues": {
      "totalCount": 12
    }
  }
]
```

3.5.2 Parsing the array

Let's deserialize this array. You should now be familiar with the process:

1. Put the JSON document in memory.
2. Compute the size with `JSON_ARRAY_SIZE()`.
3. Allocate the `JsonDocument`.
4. Call `deserializeJson()`.

Let's do it:

```
// Put the JSON input in memory (shortened)
char input[] = "[{\"name\":\"ArduinoJson\",\"stargazers\":...";

// Compute the required size
const int capacity = JSON_ARRAY_SIZE(2)
                   + 2*JSON_OBJECT_SIZE(3)
                   + 4*JSON_OBJECT_SIZE(1);

// Allocate the JsonDocument
StaticJsonDocument<capacity> doc;

// Parse the JSON input
DeserializationError err = deserializeJson(doc, input);

// Parse succeeded?
if (err) {
  Serial.print(F("deserializeJson() returned "));
  Serial.println(err.f_str());
  return;
}
```

As said earlier, a hard-coded input like that would never happen in production code, but it's a good step for your learning process.

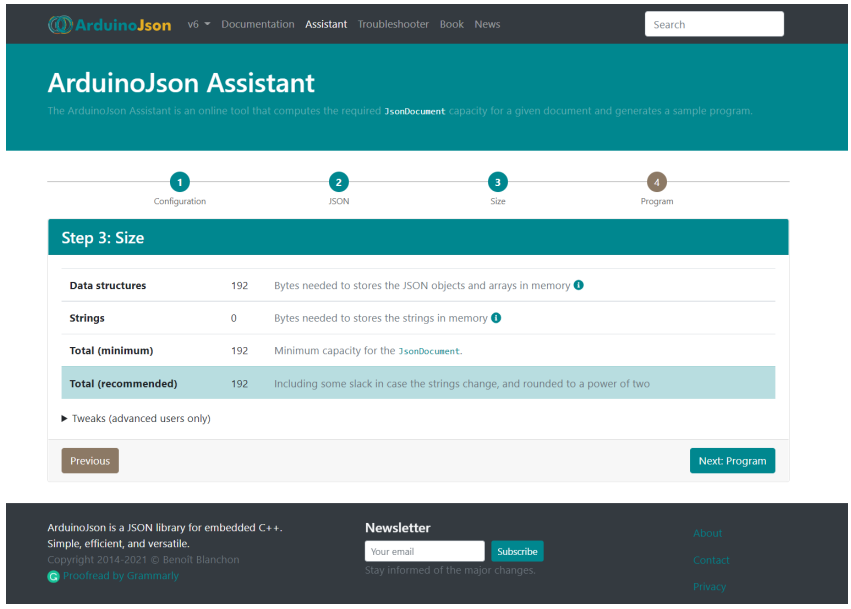
You can see that the expression for computing the capacity of the `JsonDocument` is quite complicated:

- There is one array of two elements: `JSON_ARRAY_SIZE(2)`
- In this array, there are two objects with three members: `2*JSON_OBJECT_SIZE(3)`
- In each object, there are two objects with one member: `4*JSON_OBJECT_SIZE(1)`

3.5.3 The ArduinoJson Assistant

For complex JSON documents, the expression to compute the capacity of the `JsonDocument` becomes impossible to write by hand. I did it above so that you understand the process, but in practice, we use a tool to do that.

This tool is the "ArduinoJson Assistant." You can use it online at `arduinojson.org/assistant`.



Using the ArduinoJson Assistant is straightforward:

- In step 1, you configure the Assistant
  - Processor: ESP8266
  - Mode: Deserialize
  - Input type: `char[]`
- In step 2, you enter the JSON document that you want to parse.
- Step 3 shows how big show the `JsonDocument` be. You can tweak the configuration and see the effect on the required size.
- Step 4 shows a sample program that extracts all values from the document.

Don't worry; the Assistant respects your privacy: it computes the expression locally in the browser; it doesn't send your JSON document to a web service.

3.6 Extracting values from an array

3.6.1 Retrieving elements by index

The process of extracting the values from an array is very similar to the one for objects. The only difference is that arrays are indexed by an integer, whereas objects are indexed by a string.

To get access to the repository information, we need to get the `JsonObject` from the `JsonDocument`, except that, this time, we'll pass an integer to the subscript operator (`[]`).

```
// Get the first element of the array
JsonObject repo0 = doc[0];

// Extract the values from the object
const char* name   = repo0["name"];
long        stars  = repo0["stargazers"]["totalCount"];
int         issues = repo0["issues"]["totalCount"];
```

Of course, we could have inlined the `repo0` variable (i.e., write `doc[0]["name"]` each time), but it would cost an extra lookup for each access to the object.

3.6.2 Alternative syntaxes

It may not be obvious, but the program above uses implicit casts. Indeed, the subscript operator (`[]`) returns a `JsonVariant` that is implicitly converted to a `JsonObject`.

Again, some programmers don't like implicit casts, that is why ArduinoJson offers an alternative syntax with `as<T>()`. For example:

```
auto repo0 = arr[0].as<JsonObject>();
```

All of this should sound very familiar because we've seen the same for objects.

3.6.3 When complex values are missing

When we learned how to extract values from an object, we saw that if a member is missing, a default value is returned (for example, `0` for an `int`). Similarly, ArduinoJson returns a default value when you use an index that is out of the range of an array.

Let's see what happens in our case:

```
// Get an object out of array's range
JsonObject repo666 = arr[666];
```

The index `666` doesn't exist in the array, so a special value is returned: a null `JsonObject`. Remember that `JsonObject` is a reference to an object stored in the `JsonDocument`. In this case, there is no object in the `JsonDocument`, so the `JsonObject` points to nothing: it's a null reference.

You can test if a reference is null by calling `isNull()`:

```
if (repo666.isNull()) ...
```

Alternatively, you can compare to `nullptr` (but not `NULL`!), like so:

```
if (repo666 == nullptr) ...
```

A null `JsonObject` evaluates to `false`, so you can check that it's not null like so:

```
if (repo666) ...
```

A null `JsonObject` looks like an empty object, except that you cannot modify it. You can safely call any function of a null `JsonObject`; it simply ignores the call and returns a default value. Here is an example:

```
// Get a member of a null JsonObject
int stars = repo666["stargazers"]["totalCount"];
// stars == 0
```

The same principles apply to null `JsonArray`, `JsonVariant`, and `JsonDocument`.

The null object design pattern

What we just saw is an implementation of the null object design pattern. Instead of returning `nullptr` when the value is missing, a placeholder is returned: the "null object." This object has no behavior, and all its methods fail. In short, this pattern saves you from constantly checking that a result is not null.

If ArduinoJson didn't implement this pattern, we could not write the following statement because any missing value would crash the program.

```
int stars = arr[0]["stargazers"]["totalCount"];
```

3.7 Inspecting an unknown array

In the previous section, our example was very straightforward because we knew that the JSON array had precisely two elements, and we knew the content of these elements. In this section, we'll see what tools are available when you don't know the content of the array.

3.7.1 Getting a reference to the array

Do you remember what we did when we wanted to enumerate the key-value pairs of an object? We began by calling `JsonDocument::as<JsonObject>()` to get a reference to the root object.

Similarly, if we want to enumerate all the elements of an array, the first thing we have to do is to get a reference to it:

```
// Get a reference to the root array
JsonArray arr = doc.as<JsonArray>();
```

Again, `JsonArray` is a reference to an array stored in the `JsonDocument`; it's not a copy of the array. When you apply changes to the `JsonArray`, they affect the `JsonDocument`.

3.7.2 Capacity of JsonDocument for an unknown input

If you know absolutely nothing about the input (which is strange), you need to determine a memory budget allowed for parsing the input. For example, you could decide that 10KB of heap memory is the maximum you accept to spend on JSON parsing.

This constraint looks terrible at first, especially if you're a desktop or server application developer; but, once you think about it, it makes complete sense. Indeed, your program will run in a loop on dedicated hardware. Since the hardware doesn't change, the amount of memory is always the same. Having an elastic capacity would just produce a larger and slower program with no additional value; it would also increase the heap fragmentation, which we must avoid at all costs.

However, most of the time, you know a lot about your JSON document. Indeed, there are usually a few possible variations in the input. For example, an array could have between zero and four elements, or an object could have an optional member. In that

case, use the ArduinoJson Assistant to compute the size of each variant and pick the biggest.

3.7.3 Number of elements in an array

The first thing you want to know about an array is the number of elements it contains. This is the role of `JsonArray::size()`:

```
// Get the number of elements in the array
int count = arr.size();
```

As the name may be confusing, let me clarify: `JsonArray::size()` returns the number of elements, not the memory consumption. If you want to know how many bytes of memory are used, call `JsonDocument::memoryUsage()`:

```
// How many bytes are used in the document
int memoryUsed = doc.memoryUsage();
```

Note that there is also a `JsonObject::size()` that returns the number of key-value pairs in an object, but it's rarely helpful.

3.7.4 Iteration

Now that you have the size of the array, you probably want to write the following code:

```
// BAD EXAMPLE, see below
for (int i=0; i<arr.size(); i++) {
  JsonObject repo = arr[i];
  const char* name = repo["name"];
  // etc.
}
```

The code above works but is terribly slow. Indeed, ArduinoJson stores arrays as linked lists, so accessing an element at a random location costs `O(n)`; in other words, it takes `n` iterations to get to the `n`th element. Moreover, the value of `JsonArray::size()` is not cached, so it needs to walk the linked list too.

That's why it is **essential** to avoid `arr[i]` and `arr.size()` in a loop. Instead, you should use the iteration feature of `JsonArray`, like so:

```
// Walk the JsonArray efficiently
for (JsonObject repo : arr) {
  const char* name = repo["name"];
  // etc.
}
```

With this syntax, the internal linked list is walked only once, and it is as fast as it gets.

I used a `JsonObject` in the loop because I knew that the array contains objects. If it's not your case, you can use a `JsonVariant` instead.

When C++11 is not available

The code above leverages a C++11 feature called "range-based for loop." If you cannot enable C++11 on your compiler, you must use the following syntax:

```
for (JsonArray::iterator it=arr.begin(); it!=arr.end(); ++it) {
  JsonObject repo = *it;
  const char* name = repo["name"];
  // etc.
}
```

3.7.5 Detecting the type of an element

We test the type of array elements the same way we did for object members: using `JsonVariant::is<T>()`.

Here is an example:

```
// Is the first element an integer?
if (arr[0].is<int>()) {
  // Yes!
  int value = arr[0];
  // ...
}
```

```
// Same in a loop
for (JsonVariant elem : arr) {
  // Is the current element an object?
  if (elem.is<JsonObject>()) {
    JsonObject obj = elem;
    // ...
  }
}
```

There is nothing new here, as it's exactly what we saw for object members.

3.8 The zero-copy mode

3.8.1 Definition

At the beginning of this chapter, we saw how to parse a JSON document from a *writable* source. Indeed, the `input` variable was a `char[]` in the stack, and therefore, it was writable. I told you that this fact would matter, and it's time to explain.

ArduinoJson behaves differently with writable inputs and read-only inputs.

When the argument passed to `deserializeJson()` is of type `char*` or `char[]`, ArduinoJson uses a mode called "zero-copy." It has this name because the parser never makes any copy of the input; instead, it stores pointers pointing inside the input buffer.

In the zero-copy mode, when a program requests the content of a string member, ArduinoJson returns a pointer to the beginning of the string in the input buffer. To make it possible, ArduinoJson inserts null-terminators at the end of each string; it is the reason why this mode requires the input to be writable.

The jsmn library

As we said at the beginning of the book, jsmn is a C library that detects the tokens in the input. The zero-copy mode is very similar to the behavior of jsmn.

This information should not be a surprise because the first version of ArduinoJson was just a C++ wrapper on top of jsmn.

3.8.2 An example

To illustrate how the zero-copy mode works, let's have a look at a concrete example. Suppose we have a JSON document that is just an array containing two strings:

```
["hip","hop"]
```

And let's says that the variable is a `char[]` at address `0x200` in memory:

```
char input[] = "[\"hip\",\"hop\"]";
// We assume: &input == 0x200
```

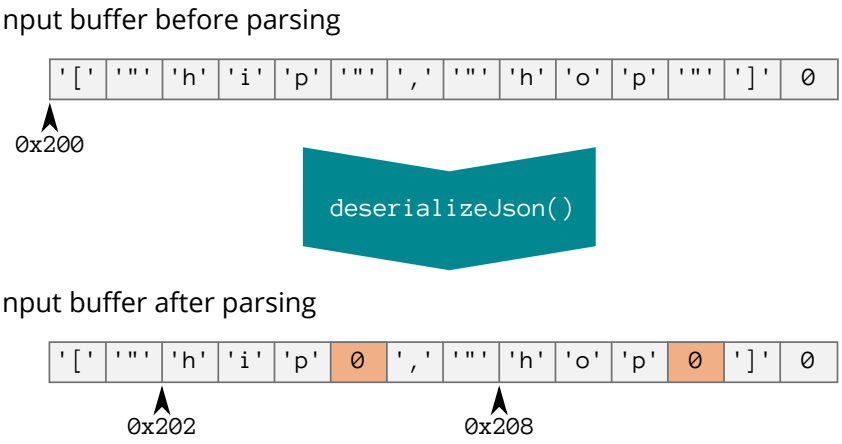
After parsing the input, when the program requests the value of the first element, ArduinoJson returns a pointer whose address is `0x202`, which is the location of the string in the input buffer:

```
deserializeJson(doc, input);

const char* hip = doc[0];
const char* hop = doc[1];
// Now: hip == 0x202 && hop == 0x208
```

We naturally expect `hip` to be `"hip"` and not `"hip\",\"hop\"]"`; that's why ArduinoJson adds a null-terminator after the first *p*. Similarly, we expect `hop` to be `"hop"` and not `"hop\"]"`, so it adds a second terminator.

The picture below summarizes this process.



Adding null-terminators is not the only thing the parser modifies in the input buffer. It also replaces escaped character sequences, like `\n`, by their corresponding ASCII characters.

I hope this explanation gives you a clear understanding of what the zero-copy mode is and why the input is modified. It is a bit of a simplified view, but the actual code is very similar.

3.8.3 Input buffer must stay in memory

As we saw, in the zero-copy mode, ArduinoJson returns pointers to the input buffer. This can only work if the input buffer is still in memory when the pointer is dereferenced.

If a program dereferences the pointer after the destruction of the input buffer, it is very likely to crash instantly, but it could also work for a while and crash later, or it could have nasty side effects. In the C++ jargon, this is what we call an "Undefined Behavior"; we'll talk about that in "Troubleshooting."

Here is an example:

```
// Declare a pointer
const char *hip;

// New scope
{
  // Declare the input in the scope
  char input[] = "[\"hip\",\"hop\"]";

  // Parse input
  deserializeJson(doc, input);
  JsonArray arr = doc.as<JsonArray>();

  // Save a pointer
  hip = arr[0];
}
// input is destructed now

// Dereference the pointer
Serial.println(hip); // <- Undefined behavior
```

Common cause of bugs

Dereferencing a pointer to a destructed variable is a common cause of bugs.

To use a `JsonArray` or a `JsonObject`, you must keep the `JsonDocument` alive. In addition, when using the zero-copy mode, you must also keep the input buffer in memory.

3.9 Reading from read-only memory

3.9.1 The example

We saw how ArduinoJson behaves with a writable input and how the zero-copy mode works. It's time to see what happens when the input is read-only.

Let's go back to our previous example except that, this time, we change its type from `char[]` to `const char*`:

```
const char* input = "[\"hip\",\"hop\"]";
```

Previously, we had the whole string duplicated in the stack, but it's not the case anymore. Instead, the stack only contains the pointer `input` pointing to the beginning of the string.

3.9.2 Duplication is required

In the zero-copy mode, ArduinoJson stores pointers pointing inside the input buffer. We saw that it has to replace some characters of the input with null-terminators.

With a read-only input, ArduinoJson cannot do that anymore, so it needs to make copies of `"hip"` and `"hop"`. Where do you think the copies would go? In the `JsonDocument`, of course!

In this mode, the `JsonDocument` holds a copy of each string, so we need to increase its capacity. Let's do the computation for our example:

1. We still need to store an object with two elements, that's `JSON_ARRAY_SIZE(2)`.
2. We have to make a copy of the string `"hip"`, that's `4` bytes, including the null-terminator.
3. We also need to copy the string `"hop"`, that's `4` bytes too.

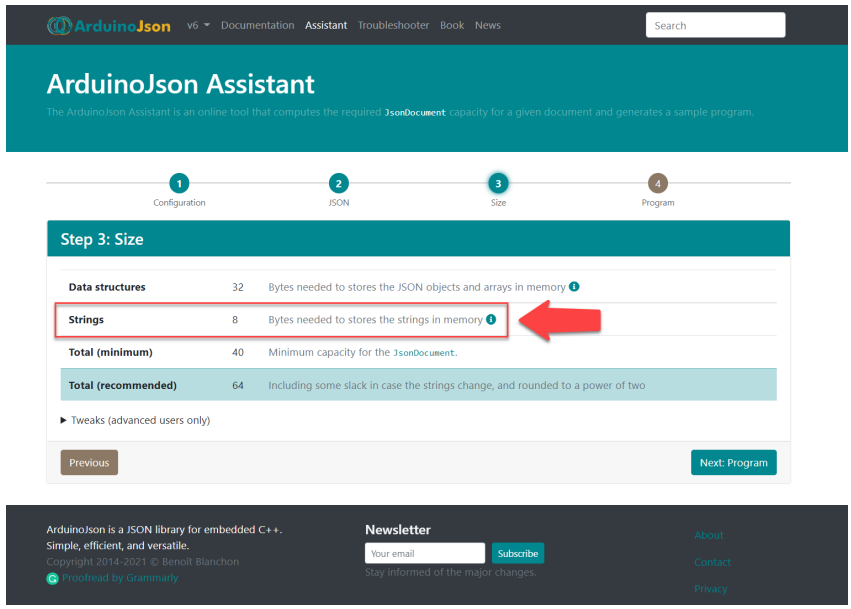
The required capacity is:

```
const int capacity = JSON_ARRAY_SIZE(2) + 8;
```

In practice, you should not use the exact length of the strings. It's safer to add a bit of slack in case the input changes. My advice is to add 10% to the longest possible string, which gives a reasonable margin.

Use the ArduinoJson Assistant

The ArduinoJson assistant also computes the number of bytes required for the duplication of strings. It shows this value in the "String" row in step 3.



3.9.3 Practice

Apart from the capacity of the `JsonDocument`, we don't need to change anything to the program.

Here is the complete hip-hop example with a read-only input:

```
// A read-only input
const char* input = "[\"hip\",\"hop\"]";

// Allocate the JsonDocument
const int capacity = JSON_ARRAY_SIZE(2) + 8;
```

```
StaticJsonDocument<capacity> doc;

// Parse the JSON input.
deserializeJson(doc, input);

// Extract the two strings.
const char* hip = doc[0];
const char* hop = doc[1];

// How much memory is used?
int memoryUsed = doc.memoryUsage();
```

I added a call to `JsonDocument::memoryUsage()`, which returns the current memory usage. Do not confuse it with the *capacity*, which is the maximum size.

If you compile this program on an ESP8266, the variable `memoryUsed` will contain `40`, as the ArduinoJson Assistant predicted.

3.9.4 Other types of read-only input

`const char*` is not the sole read-only input that ArduinoJson supports. For example, you can also use a `String`:

```
// Put the JSON input in a String
String input = "[\"hip\",\"hop\"]";
```

It's also possible to use a Flash string, but there is one caveat. As we said in the C++ course, ArduinoJson needs a way to figure out if the input string is in RAM or Flash. To do that, it expects a Flash string to have the type `const __FlashStringHelper*`. If you declare a `char[] PROGMEM`, ArduinoJson will not consider it as Flash string, unless you cast it to `const __FlashStringHelper*`.

Alternatively, you can use the `F()` macro, which casts the pointer to the right type:

```
// Put the JSON input in the Flash
auto input = F("[\"hip\",\"hop\"]");
// (auto is deduced to const __FlashStringHelper*)
```

As we saw in the previous chapter, using `F()` and `PROGMEM` strings only makes sense on Harvard architectures, such as AVR and ESP8266.

In the next section, we'll see another kind of read-only input: streams.

3.10 Reading from a stream

In the Arduino jargon, a stream is a volatile source of data, like the serial port or a TCP connection. Contrary to a memory buffer, which allows reading any bytes at any location (after all, that's what the acronym "RAM" means), a stream only allows reading one byte at a time and cannot rewind.

The `Stream` abstract class materializes this concept. Here are examples of classes derived from `Stream`:

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

`std::istream`

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

ArduinoJson can use both `Stream` and `std::istream`.

3.10.1 Reading from a file

As an example, we'll create a program that reads a JSON file stored on an SD card. We suppose that this file contains the array we used as an example earlier.

The program will just read the file and print the information for each repository.

Here is the relevant part of the code:

```
// Open file
File file = SD.open("repos.txt");

// Parse directly from file
deserializeJson(doc, file);

// Loop through all the elements of the array
for (JsonObject repo : doc.as<JsonArray>()) {
  // Print the name, the number of stars, and the number of issues
  Serial.println(repo["name"].as<const char*>());
  Serial.println(repo["stargazers"]["totalCount"].as<int>());
  Serial.println(repo["issues"]["totalCount"].as<int>());
}
```

A few things to note:

1. I used the `.txt` extension instead of `.json` because the FAT file system is limited to three characters for the file extension.
2. I used the ArduinoJson Assistant to compute the capacity (not shown above because it's not the focus of this snippet).
3. I called `JsonVariant::as<T>()` to pick the right overload of `Serial.println()`.

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

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

3.10.2 Reading from an HTTP response

Now is the time to parse the actual data coming from GitHub's API!

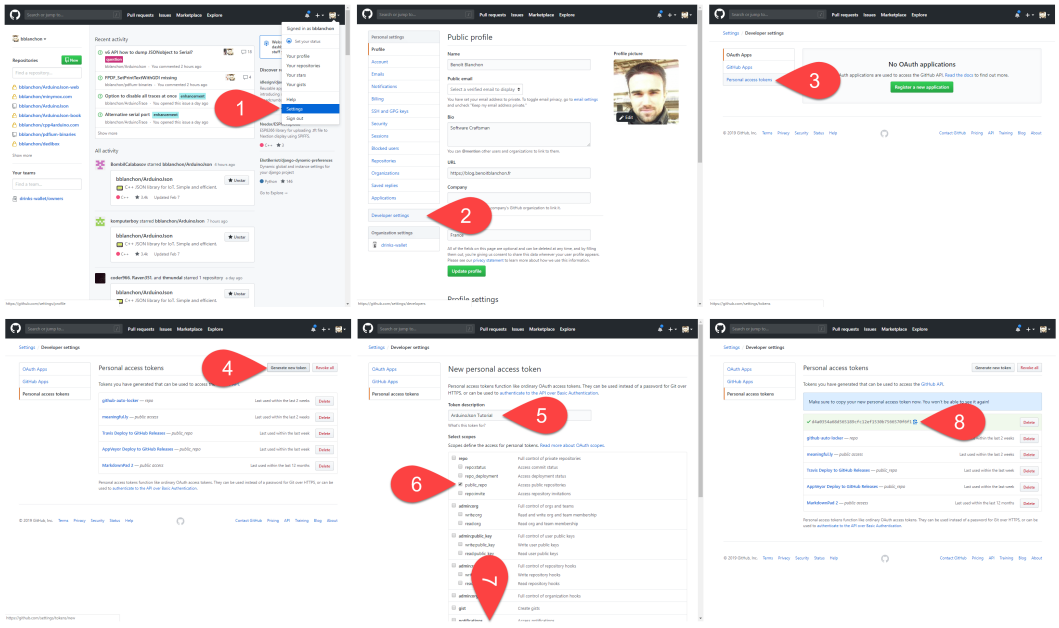
As I said, we need a microcontroller that supports HTTPS, so we'll use an ESP8266 with the library "ESP8266HTTPClient." Don't worry if you don't have a compatible board; we'll see other configurations in the case studies.

Access token

Before using this API, you need a GitHub account and a "personal access token." This token grants access to the GitHub API from your program; we might also call it an "API key." To create it, open GitHub in your browser and follow these steps:

1. Go to your personal settings.
2. Go in "Developer settings."
3. Go in "Personal access token."
4. Click on "Generate a new token."
5. Enter a name, like "ArduinoJson tutorial."
6. Check the scopes (i.e., the permissions); we only need "public_repo."
7. Click on "Generate token."
8. GitHub shows the token.

You can see each step in the picture below:



GitHub won't show the token again, so don't waste any second and write it in the source code:

```
#define GITHUB_TOKEN "d4a0354a68d565189cfc12ef1530b7566570f6f1"
```

With this token, our program can authenticate with GitHub's API. All we need to do is to add the following HTTP header to each request:

```
Authorization: bearer d4a0354a68d565189cfc12ef1530b7566570f6f1
```

Certificate validation

Because I don't want to make this example more complicated than necessary, I'll disable the SSL certificate validation, like so:

```
WiFiClientSecure client;
client.setInsecure();
```

What could be the consequence? Since the program doesn't verify the certificate, it cannot be sure of the server's authenticity, so it could connect to a rogue serve that pretends to be `api.github.com`. This is indeed a serious security breach because the program would send your Personal Access Token to the rogue server. Fortunately, this token has minimal permissions: it only provides access to public information. However, in a different project, the consequences could be disastrous.

If your project presents any security or privacy risk, you must enable SSL certificate validation. `WiFiClientSecure` provides several validation methods. For a simple solution, use `setFingerprint()`, but you'll have to update the fingerprint frequently. For a more robust solution, use `setTrustAnchors()` and make sure your clock is set to the current time and date.

The request

To interact with the new GraphQL API, we need to send a `POST` request (instead of the more common `GET` request) to the URL `https://api.github.com/graphql`.

The body of the `POST` request is a JSON object that contains one string named "query." This string contains a GraphQL query. For example, if we want to get the name of the

authenticated user, we need to send the following JSON document in the body of the request:

```
{
  "query": "{viewer{name}}"
}
```

The GraphQL syntax and the details of GitHub's API are obviously out of the scope of this book, so I'll simply say that a GraphQL query allows you to select the information you want within the universe of information that the API exposes.

In our case, we want to retrieve the names, numbers of stars, and numbers of opened issues of your ten most popular repositories. Here is the corresponding GraphQL query:

```
{
  viewer {
    name
    repositories(ownerAffiliations: OWNER,
                 orderBy: {
                   direction: DESC,
                   field: STARGAZERS
                 },
                 first: 10) {
      nodes {
        name
        stargazers {
          totalCount
        }
        issues(states: OPEN) {
          totalCount
        }
      }
    }
  }
}
```

To find the correct query, I used the GraphQL API Explorer. With this tool, you can test GraphQL queries in your browser. You can find it in GitHub's API documentation.

We'll reduce this query to a single line to save some space and bandwidth; then, we'll put it in the "query" string in the JSON object. Since we haven't talked about JSON

serialization yet, we'll hard-code the string in the program.

To summarize, here is how we will send the request:

```
HTTPClient http;
http.begin(client, "https://api.github.com/graphql");
http.addHeader("Authorization", "bearer " GITHUB_TOKEN));
http.POST("{\"query\":\"{viewer{name,repositories(ownerAffiliations:...");
```

The response

After sending the request, we must get a reference to the `Stream`:

```
// Get a reference to the stream in HTTPClient
Stream& response = http.getStream();
```

As you see, we call `getStream()` to get the internal stream (we could have used `client` directly). Unfortunately, when we do that, we bypass the part of ESP8266HTTPClient that handles chunked transfer encoding. To make sure GitHub doesn't return a chunked response, we must set the protocol to HTTP 1.0:

```
// Downgrade to HTTP 1.0 to prevent chunked transfer encoding
http.useHTTP10(true);
```

Because the protocol version is part of the request, we must call `useHTTP10()` before calling `POST()`.

Now that we have the stream, we can pass it to `deserializeJson()`:

```
// Allocate the JsonDocument in the heap
DynamicJsonDocument doc(2048);

// Deserialize the JSON document in the response
deserializeJson(doc, response);
```

Here, we used a `DynamicJsonDocument` because it is too big for the stack. As usual, I used the ArduinoJson Assistant to compute the capacity.

The body contains the JSON document that we want to deserialize. It's a little more complicated than what we saw earlier. Indeed, the JSON array is not at the root but under `data.viewer.repositories.nodes`, as you can see below:

```
{
  "data": {
    "viewer": {
      "name": "Benoît Blanchon",
      "repositories": {
        "nodes": [
          {
            "name": "ArduinoJson",
            "stargazers": {
              "totalCount": 5246
            },
            "issues": {
              "totalCount": 15
            }
          },
          {
            "name": "pdfium-binaries",
            "stargazers": {
              "totalCount": 249
            },
            "issues": {
              "totalCount": 12
            }
          },
          ...
        ]
      }
    }
  }
}
```

So, compared to what we saw earlier, the only difference is that we'll have to walk several objects before getting the reference to the array. The following line will do:

```
JsonArray repos = doc["data"]["viewer"]["repositories"]["nodes"];
```

The code

I think we have all the pieces, let's assemble this puzzle:

```
// Prepare the WiFi client
WiFiClientSecure client;
client.setInsecure();

// Send the request
HTTPClient http;
http.begin(client, "https://api.github.com/graphql");
http.useHTTP10(true);
http.addHeader("Authorization", "bearer " GITHUB_TOKEN));
http.POST("{\"query\":\"{viewer{name,repositories(ownerAffiliations:...");

// Get a reference to the stream in HTTPClient
Stream& response = http.getStream();

// Allocate the JsonDocument in the heap
DynamicJsonDocument doc(2048);

// Deserialize the JSON document in the response
deserializeJson(doc, response);

// Get a reference to the array
JsonArray repos = doc["data"]["viewer"]["repositories"]["nodes"];

// Print the values
for (JsonObject repo : repos) {
  Serial.print(" - ");
  Serial.print(repo["name"].as<char *>());
  Serial.print(", stars: ");
  Serial.print(repo["stargazers"]["totalCount"].as<long>());
  Serial.print(", issues: ");
  Serial.println(repo["issues"]["totalCount"].as<int>());
}

// Disconnect
http.end();
```

If all works well, this program should print something like so:

```
- ArduinoJson, stars: 5246, issues: 15
- pdfium-binaries, stars: 249, issues: 12
- ArduinoStreamUtils, stars: 131, issues: 3
- ArduinoTrace, stars: 125, issues: 1
- WpfBindingErrors, stars: 77, issues: 4
- dllhelper, stars: 27, issues: 0
- cpp4arduino, stars: 26, issues: 1
- HighSpeedMvvm, stars: 15, issues: 0
- SublimeText-HighlightBuildErrors, stars: 12, issues: 4
- BuckOperator, stars: 10, issues: 0
```

You can find the complete source code of this example in the `GitHub` folder in the zip file provided with the book. Compared to what is shown above, the source code handles the connection to the WiFi network, check errors, and uses Flash strings when possible.

3.11 Summary

In this chapter, we learned how to deserialize a JSON input with ArduinoJson. Here are the key points to remember:

- `JsonDocument`:
  - `JsonDocument` stores the memory representation of the document.
  - `StaticJsonDocument` is a `JsonDocument` that resides in the stack.
  - `DynamicJsonDocument` is a `JsonDocument` that resides in the heap.
  - `JsonDocument` has a fixed capacity that you set at construction.
  - You can use the ArduinoJson Assistant to compute the capacity.
- `JsonArray` and `JsonObject`:
  - You can extract the value directly from the `JsonDocument` as long as there is no ambiguity.
  - To solve an ambiguity, you must call `as<JsonArray>()` or `as<JsonObject>()`.
  - `JsonArray` and `JsonObject` are references, not copies.
  - The `JsonDocument` must remain in memory; otherwise, the `JsonArray` or the `JsonObject` contains a dangling pointer.
- `JsonVariant`:
  - `JsonVariant` is also a reference and supports several types: object, array, integer, float, and boolean.
  - `JsonVariant` differs from `JsonDocument` because it doesn't own the memory; it just points to it.
  - `JsonVariant` supports implicit conversion, but you can also call `as<T>()`.
- The two modes:
  - The parser has two modes: zero-copy and classic.
  - It uses the zero-copy mode when the input is a `char*`.
  - It uses the classic mode with all other types.

  - The zero-copy mode allows having a smaller `JsonDocument` because it stores pointers to the strings in the input buffer.

In the next chapter, we'll see how to serialize a JSON document with ArduinoJson.

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**.