Universal Binary JSON Documentation

Release 0.8

UBJSON Community

August 21, 2014

Contents

JSON has become a ubiquitous text-based file format for data interchange. Its simplicity, ease of processing and (relatively) rich data typing made it a natural choice for many developers needing to store or shuffle data between systems quickly and easy.

Unfortunately, marshaling native programming language constructs in and out of a text-based representations does have a measurable processing cost associated with it.

In high-performance applications, avoiding the text-processing step of JSON can net big wins in both processing time and size reduction of stored information, which is where a binary JSON format becomes helpful.

CHAPTER 1

Specification

1.1 Data Format

The Universal Binary JSON specification utilizes a single binary tuple to represent all JSON data types (both value and container types):

```
<type, 1-byte char>[<length, 1 or 4-byte integer>][<data>]
```

Each element in the tuple is defined as:

- **type**
  - A 1-byte ASCII char used to indicate the type of the data following it.
  - A single ASCII char was chosen to make manually walking and debugging data stored in the Universal Binary JSON format as easy as possible (e.g. making the data relatively readable in a hex editor).
- **length** (OPTIONAL) 1-byte or 4-byte length value based on the type specified. This allows for more aggressive compression and space-optimization when dealing with a lot of small values.
  - 1-byte: An unsigned byte value (`0` to `254`) used to indicate the length of the data payload following it. Useful for small items.
  - 4-byte: An unsigned integer value (`0` to `2,147,483,647`) used to indicate the length of the data payload following it. Useful for larger items.
- **data** (OPTIONAL) A run of bytes representing the actual binary data for this type of value.

In the name of efficiency, the length and data fields are optional depending on the type of value being encoded. Some value are simple enough that just writing the 1-byte ASCII marker into the stream is enough to represent the value (e.g. *null*) while others have a type that is specific enough that no length is needed as the length is implied by the type (e.g. *int32*).

The specifics of each data type will be spelled out down below for more clarity.

The basic organization provided by this tuple (*type-length-data*) allows each JSON construct to be represented in a binary format that is simple to read and write without the need for complex/custom encodings or `null`-termating bytes anywhere in the stream that has to be scanned for or references resolved.

1.2 Value Types

This section describes the mapping between the 5 discrete value types from the JSON specification into the Universal Binary JSON format.

1.2.1 JSON

The JSON specification defines 7 value types:

- string
- number
- object (container)
- array (container)
- true
- false
- null

Of those 7 values, 2 of them are types describing containers that hold the 5 basic values. We have a separate section below for looking at the 2 container types specifically, so for the time being let's only consider the following 5 discrete value types:

- string
- number
- true
- false
- null

Most of these types have a `1:1` mapping to a primitive type in most popular programming languages (Java, C, Python, PHP, Erlang, etc.) except for *number*. This makes defining the types for the 4 easy, but let's take a closer look at how we might deconstruct *number* into its core representations.

Number Type

In JavaScript, the Number type can represent any numeric value where as many other languages define numbers using 3-6 discrete numeric types depending on the type and length of the value being stored. This allows the runtime to handle numeric operations more efficiently.

In order for the Universal Binary JSON specification to be a performant alternative to JSON, support for these most common numeric types had to be added to allow for more efficient reading and writing of numeric values.

number is deconstructed in the Universal Binary JSON specification and defined by the following **signed** numeric types:

- byte (8-bits, 1-byte)
- int16 (16-bits, 2-bytes)
- int32 (32-bits, 4-bytes)
- int64 (64-bits, 8-bytes)
- float (32-bits, 4-bytes)
- double (64-bits, 8-bytes)
- huge (arbitrarily long, UTF-8 string-encoded numeric value)

Trying to maintain a single *number* type represented in binary form would have lead to parsing complexity and slow-downs as the processing language would have to further inspect the value and map it to the most optimal type. By

pre-defining these different numeric types directly in binary, in most languages the number can stay in their optimal form on disk and be deserialized back into their native representation with very little overhead.

When working on a platform like JavaScript that has a singular type for numbers, all of these data types (with the exception of *huge*) can simply be mapped back to the *number* type with ease and no loss of precision.

When converting these formats back to JSON, all of the numeric types can simply be rendered as the singular number type defined by the JSON spec without issue; there is total compatibility!

Value Type Summary

Now that we have clearly defined all of our (signed) numeric types and mapped the 4 remaining simple types to Universal Binary JSON, we have our final list of discrete value types:

- null
- false
- true
- byte
- int16
- int32
- int64
- float
- double
- huge
- string

Now that we have defined all the types we need, let's see how these are actually represented in binary in the next section.

1.2.2 Universal Binary JSON

The Universal Binary JSON specification defines a total of 13 discrete value types (that we saw in the last section) all delimited in the binary file by a specific, 1-byte ASCII character (optionally) followed by a length and (optionally) a data payload containing the value data itself.

Some of the values (*null*, *true*‘ and *false*) are specific enough that the single 1-byte ASCII character is enough to represent the value in the format and they will have no *length* or *data* section.

All of the numeric values (except *huge*) automatically imply a length by virtue of the type of number they are. For example, a 4-byte *int32* always has a length of 4-bytes; an 8-byte *double* always requires 8 bytes of data.

In these cases the ASCII marker for these types are immediately followed by the data representing the number with no *length* value in between.

Because *string* and *huge* are potentially variable length, they contain all 3 elements of the tuple: *type-length-data*.

This table shows the official definitions of the discrete value types:

Type	Size	Marker	Length?	Data?
null	1-byte	Z	No	No
true	1-byte	T	No	No
false	1-byte	F	No	No
byte	2-bytes	B	No	Yes
int16	3-bytes	i	No	Yes
int32	5-bytes	I	No	Yes
int64	9-bytes	L	No	Yes
float (32-bit)	5-bytes	d	No	Yes
double (64-bit)	9-bytes	D	No	Yes
huge (number)	2-bytes + byte length of string	h	Yes	Yes if non-empty
huge (number)	5-bytes + byte length of string	H	Yes	Yes, if non-empty
string	2-bytes + byte length of string	s	Yes	Yes, if non-empty
string	5-bytes + byte length of string	S	Yes	Yes, if non-empty

Note: The duplicate (lowercased) `h` and `s` types are just versions of those types that allow for a 1-byte length (instead of 4-byte length) to be used for more compact storage when length is `<= 254`.

With each field of the table described as:

- **Type**
  - The binary value data type defined by the spec.
- **Size**
  - The byte-size of the construct, as stored in the binary format. This is not the value of the *length* field, just an indicator to you (the reader) of approximately how much space writing out a value of this type will take.
- **Marker**
  - The single ASCII character marker used to delimit the different types of values in the binary format. When reading in bytes from a file stored in this format, you can simply check the decimal value of the byte (e.g. `'A' = 65`) and switch on that value for processing.
- **Length?**
  - Indicates if the data type provides a length value between the ASCII marker and the data payload.
  - Many of the data types represented in the binary format either don't have a length (*null*, *true* or *false*) or their types (e.g. the numeric values) are specific enough that the length is implied.
  - When specifying the length for a string or huge value (UTF-8 encoded), the length **must represent the number of bytes** of the UTF-8 string and not the number of characters in the string.

  Note: For example, English typically uses 1-byte per character, so the string "hello" has a length of 5. The same string in Russian is "" with a byte length of 12 and in Arabic the text becomes "" with a byte length of 10.
- **Data?**
  - Indicates if the data type provides a data payload representing the value.
  - Most types except for *null*, *true* and *false* provide a data payload indicating their value.
  - Variable-length types like *string* and *huge* **do not** provide a data payload when they are empty (i.e. length of 0).More specifically, if you are writing a parser for the Universal Binary JSON format and you encounter a *string* of length 0, you know the very next byte is an ASCII marker for another value since the *string* has no data payload.

Note: Using Numeric Types

It is always recommended to use the smallest numeric type that fits your needs. For data with a large amount of numeric data, this can cut down the size of the payloads significantly (on average a **50% reduction in size**).

All numeric types are **signed**.

Numeric values of *infinity* are encoded as a *null* (`Z`) value. (See ECMA, See JSON presentation)

64-bit Integers

While almost all languages native support 64-bit integers, not all do (e.g. C89 and JavaScript (yet)) and care must be taken when encoding 64-bit integer values into binary JSON then attempting to decode it on a platform that doesn't support it.

If you are fully aware of the platforms and runtime environments your binary JSON is being used on and know they all support 64-bit integers, then you are fine.

If you are trying to deserialize 64-bit integers in a client's browser in JavaScript or another environment that does not support 64-bit integers, then you will want to take care to skip them in the input or have the client producing them encode them as *double* or *huge* values if that is easier to handle.

Alternatively you might consider encoding your 64-bit values as doubles if you know you are going from the server to a client JavaScript environment with the binary-encoded information.

32-bit Floats

All 32-bit float values are written into the binary format using the IEEE 754 single precision floating point format, which is the following structure:

- Bit 31 (1 bit) – sign
- Bit 30-23 (8 bits) – exponent
- Bit 22-0 (23 bits) – fraction (significand)

64-bit Doubles

All 64-bit double values are written into the binary format using the IEEE 754 double precision floating point format, which is the following structure:

- Bit 63 (1 bit) – sign
- Bit 62-52 (11 bits) – exponent
- Bit 51-0 (52 bits) – fraction (significand)

huge Numeric Type

The huge numeric type is a safe and portable way for representing **values > 64-bit** by way of an UTF-8 encoded string. The format of this string **must adhere** to the JSON number specification.

This allows *huge* numbers to be portable across all platforms and easily converted to/from JSON as well as more robust handling on platforms that may not support arbitrarily large numbers.

If you are working on a platform that has no support for huge numbers, please see our *Best Practices* recommendation on how to handle them.

It is considered a violation of this specification to store numeric **values <= 64-bit** as a *huge*.

This decision was made in order to simplify the parsing logic required to process the Universal Binary JSON specification; there is no need to introspect *huge* values to see if they contain smaller numeric values when mapping UBJSON types to native types of the runtime environment.

The *huge* type should only be used when you need to (safely and portably) represent **values > 64-bit**.

UTF-8 Encoding

The JSON specification does not dictate a specific required encoding, it does however use UTF-8 as the default encoding.

The Universal Binary JSON specification dictates UTF-8 as the **required string encoding**. This will allow you to easily exchange binary JSON between open systems that all follow this encoding requirement.

Fortunately, this is ideal for a multitude of reasons like space efficiency and compatibility across systems and alternative formats.

To further clarify the binary layout of these data types, below are some visual examples of what the bytes would look like inside of a binary JSON file.

NOTE: `[ ]`-block notation is used for readability, the `[ ]` characters **are not** actually written out in the binary format.

Binary Representation	Description
`[Z]`	1-byte, null value
`[T]`	1-byte, true value
`[F]`	1-byte, false value
`[B][127]`	2-bytes, 8-bit byte value of 127
`[I][32427]`	5-bytes, 32-bit integer value of 32,427
`[L][12147483647]`	9-bytes, 64-bit integer value of 12,147,483,647
`[d][3.14159]`	5-bytes, 32-bit float value of 3.14159
`[D][72.38138221]`	9-bytes, 64-bit double value of 72.38138221
`[s][5][hello]`	7 bytes, string UTF-8 “hello” (English)
`[s][12][]`	14 bytes, string UTF-8 “hello” (Russian)
`[s][10][]`	12 bytes, string UTF-8 “hello” (Arabic)
`[S][1024][...long string...]`	5 bytes + 1024 bytes for the long string
`[s][4][name][s][3][bob]`	6 + 5 bytes, equivalent of “name”: “bob”

Now that we have seen how the JSON data value types map to the binary format, in the next section we will see how we can combine these values together into the two container types (objects and arrays) to create complex object hierarchies using the Universal Binary JSON format.

1.3 Container Types

In this section we will look at the 2 remaining JSON value types that we are referring to as “container types”, namely object and array.

1.3.1 JSON

The two JSON container types are described as follows:

- **object**
  - A construct containing 0 or more name-value pairings, where the name is always a string and the value can be any valid value type including container types themselves.
- **array**
  - A flat list of values only, where the values can by any valid value type including container types themselves.
  - The JSON specification does not make it a requirement that the values in an array are all of the same type and neither does the Universal Binary JSON specification.

Note: **Advanced**: This can actually be to your benefit. Take for example an array of *int64* values, as you are writing

them out to a file or a stream, you can check the actual value of each *int64* and depending on the value, encode each one into the smallest possible numeric type (e.g. *byte*, *int32*, etc.).

This can lead to a significant size reduction (say **50% smaller**) in smaller numeric values!

Given these two constructs, let's see how they are modeled in the Universal Binary JSON format.

1.3.2 Universal Binary JSON

The two container types defined by JSON are modeled using the same tuple that defines all of our other data structures in this specification so far with a minor modification: the length value is considered a count of the child elements the container holds. It does not mean the byte length of the child elements.

Note: Exactly what *child element* means depends on the container. In an *object*, a single child element is a name-value pair; in an *array*, a child element is a single value.

More specifically, the tuple stays exactly the same, it is just the meaning of the center *length* element that changes:

```
<type, 1-byte char>[<length, 1 or 4-byte integer>][<data>]
```

All the code used to process the constructs defined by this specification stays the same, but when an *object* or *array* construct are encountered, the code needs to be aware that the *length* value is the **child element count** so it can know when the scope of the container ends.

For example, if you have an object that contains 4 arrays of *length* 50, the *length* argument for the object is 4 (because it contains the four arrays) while the *length* argument for each array is 50 (because they each hold 50 elements).

Note: Unknown-length container types are also supported by the Universal Binary JSON specification and are covered in detail in the *Streaming* section of this document.

Additionally, the only optional field in the tuple for container types is *data*, if the container is empty and contains no elements (i.e. the *length* is 0) then there is no *data* segment.

All together, the definitions for the *object* and *array* container types looks like this:

Type	Size	Marker	Length?	Data?
array	2-bytes + byte length of string	a	Yes	Yes, if non-empty
array	5-bytes + byte length of string	A	Yes	Yes, if non-empty
object	2-bytes + byte length of string	o	Yes	Yes if non-empty
object	5-bytes + byte length of string	O	Yes	Yes, if non-empty

---

Note: *array* and *object* can also be specified in a more compact manner using 1-byte for the *length* when it is <= `254`. Specifying a *length* of `255` for the 1-byte variants has a special meaning of **length unknown** and is covered in more detail in the *Streaming* section of the spec.

The details for each field are the same as described for the non-container values in the previous section with the one caveat that *length* is a count of child elements and **not** the number of bytes representing the contents of the container.

Let's look at a quick example of encoding an object, again using the handy `[ ]`-notation we used before simply for readability (the `[ ]` chars are not written out or part of the file format).

Consider the following JSON (30-bytes compacted):

```
{
    "id": 1234567890,
    "name": "bob"
}
```

Storing that object in the Universal Binary JSON format would look like this (whitespace added for readability):

```
[o][2] 2 bytes
  [s][2][id][I][1234567890] 4 + 5 = 9 bytes
  [s][4][name][s][3][bob]   6 + 5 = 11 bytes
```

Our Universal Binary JSON format is 22 bytes, **27% smaller** than our compacted JSON!

Walking through our example above, using a word-journey this is what a parser might see and do:

1. I see an `o`, so I know I am parsing an *object* and that the next byte is the *length* (or count) for this object.
2. I see a `2`, so I know the object contains 2 elements that I must account for to know when the *object* scope is closed (because we don't use the `{ }` brackets like JSON).
3. I see an `s`, knowing how the name-value pairings inside of an object work, I know this is the *name* portion of some upcoming value.
4. I see an `I`, I know this is an *int32* value and that it belongs to the *name* I parsed in the previous step.
5. I see another `s`, I know this is a new name-value pair and this is the *name* portion.
6. I see another `s` and know this is the value belonging to the *name* I just processed.
7. I have just parsed 2 values, so now I know the *object* scope is closed.

Encoding objects containing other *objects* would work identically except we would have encountered another `o` or `O` marker and descended a level further into a new object.

Let's look at another example, this time a simple JSON array construct (remember, they only contain values and not name-value pairs like *objects*).

This array is 48-bytes in compacted JSON:

```
[
  null,
  true,
  false,
  4782345193,
  153.132417549,
  "ham"
]
```

Storing the array in the Universal Binary JSON format would look like this (whitespace added for readability):

```
[a][6] - 2 bytes
   [Z] - 1 byte
   [T] - 1 byte
   [F] - 1 byte
   [I][4782345193] - 5 bytes
   [D][153.132417549] - 9 bytes
   [s][3][ham] - 5 bytes
```

Our Universal Binary JSON format is 24 bytes or **50% smaller** than the compacted JSON!

Because the container types specify their total child element count, it is easier and faster for parsers to know when the scope of a container has closed or is still open waiting for more children (e.g. in the case of streaming over the network). This is not unlike the high-performance Redis protocol.

This also has the added benefit of not needing any terminating values in the binary that need to be scanned for to know when a container-scope is closed. This way data can be read in chunks and not read-and-scanned byte-by-byte.

As was mentioned previously though, there are some cases where having an unbounded container are important (for example, streaming content from a server as it generates it on-the-fly).

In the next section we will take a look at the Universal Binary JSON constructs that are optimized for streaming. Fortunately, there are only 3 and they are just as easy as the constructs we have covered so far!

1.4 Streaming Types

The Universal Binary JSON specification is optimized for fast read-speed by prefixing the byte-length of every construct to the front of it, this allows parsers to digest entire chunks of the data stream at a time without scanning for terminating byte values.

Unfortunately, this model of data becomes very expensive (and sometimes impossible) to adhere to in a streaming-friendly environment where a server may be generating *UBJ* formatted data on-the-fly and streaming it back in real time to the client.

If the server had to adhere to the prefixed-length requirement of this specification up until now, it would have to generate, buffer and count all the elements in its reply before writing out the Universal Binary JSON so it could correctly prefix the lengths to all the containers.

In this section of the specification we look at 1 new additional type to the Universal Binary JSON specification that compliments our streaming scenario and then two minor changes to the existing **container types** to enable easy and efficient streaming with unknown-length support for our *array* and *object* containers.

1.4.1 No-Op Type

The noop value stands for *No Op* or *No Operation*, it is a specific value (like `Z` for *null*, `T` for *true* and `F` for *false*) that is useful in streaming scenarios where an acknowledge of life needs to be sent between two end points, but the confirmation being sent cannot change the meaning of the data it is sent within.

The most common use for such a value type is as a *keep-alive* signal from a server to the client; letting the client know the server is possibly operating on a long-running job and is still alive, but just isn't ready to send more data yet.

The *noop* type is defined as follows:

Type	Size	Marker	Length?	Data?
noop	1-byte	N	No	No

Any parser code written to load the Universal Binary Spec needs to be aware that encountering the `N` marker in files of any kind is valid and is merely useful as a signal mechanism from producer to consumer to say "Hey, I am still alive.", the marker is intended to be safely ignored if the server or client doesn't need the acknowledgement.

In order for this keep-alive-esque construct to work, the specification had to define a single byte value that had no meaning for the server and client to exchange if needed, but caused no modification to the meaning of the data that they are exchanging.

In code that handles streaming from a server, your handler for the *noop* type might just reset a disconnect timer. In code that handles UBJ files, you would simply ignore the noop marker when you encountered it in the file because it would mean nothing.

> **Warning:** The *noop* type is only defined to be used inside of an *unknown-length container*. If you have a container that clearly defines a child element count (*length*) it should not contain a *noop* marker element.
> Also, the *noop* type **should never** be sent inside of a value (e.g. embedded inside of a *string* being streamed); it must only be written to the stream between declared values.

If your interaction with the Universal Binary JSON format is primarily as a file format, it is likely that you may never need to use the *noop* type; its value becomes more apparent in long-lived, client-server, data-streaming scenarios.

1.4.2 Unknown-Length Containers

The Universal Binary JSON specification supports containers (*array* and *object*) of unknown length to be specified when the producer of the binary data cannot (efficiently) know in advance how many elements it is going to write out.

In these cases, the lowercased, 1-byte-length versions of array or object must be used (`a` or `o` markers) with a length value of `0xFF` (255) as well as specifying an `E` terminator character after the last element in the container.

The `E` type used to delimit the end of unknown-length containers is defined as follows:

Type	Size	Marker	Length?	Data?
end	1-byte	E	No	No

Warning: Using a length of `0xFF` with the uppercase, 4-byte-length versions of array (`A`) and object (`O`) is not valid according to this specification. You must use the 1-byte-length variants of the container types when specifying an unknown *length*.

An example would look like this:

```
[a][255]
   [S][3][bob]
   [I][1024]
   [T]
   [F]
   [S][4][ham!]
[E]
```

The three key elements being the lowercased `a` marker, the length of `0xFF` (255) and the `E` marker at the end of the container.

Another example might look like this:

```
[o][255]
   [B][4]
   [D][21.786]
   [N]
   [Z]
   [h][27][131.098412283059e2371293452]
[E]
```

You might notice in the example above we injected a *noop* instruction right in the middle, before the *null*. As mentioned in the *No-Op Type* section, this is valid and can occur at any time while parsing the contents of an *unknown-length* container.

If your parser has no need for recognizing the *noop* code (e.g. listening for a keep-alive) then it can just be safely ignored and parsing continued (hence the name "no-op"). It is up to the implementation to decide what to do with the *noop* type.

You might be wondering how using a 1-byte `E` as a terminator to an unbounded container can work and not get confused with say another `E` inside of a *string*, the reason this works is because none of the discrete value types (numeric, string, etc.) are of unknown *length*.

The lengths of all the values contained inside of the container are known and must be read completely, doing so will guarantee that the `E` is only ever encountered by itself as an element marker which is easily handled by parsing code to know the scope of the container has been closed.

1.5 Size Requirements

The Universal Binary JSON specification tries to strike the perfect balance between space savings, simplicity and performance.

Data stored using the Universal Binary JSON format are on average **30% smaller** as a rule of thumb. As you can see from some of the examples in this document though, it is not uncommon to see the binary representation of some data lead to a **50% or 60% reduction in size**.

The size reduction of your data depends heavily on the type of data you are storing. It is best to do your own benchmarking with a comprehensive sampling of your own data.

Warning: The Universal Binary JSON specification does not use compression algorithms to achieve smaller storage sizes. The size reduction is a side effect of the efficient binary storage format.

1.5.1 Size Reduction Tips

The amount of storage size reduction you'll experience with the Universal Binary JSON format will depend heavily on the type of data you are encoding.

Some data shrinks considerably, some mildly and some not at all, but in every case your data will be stored in a much more efficient format that is faster to read and write.

Below are pointers to give you an idea of how certain data may shrink in this format:

- *null*, *true* and *false* values will compress 75% (80% in the case of *false*)
- large *numeric* values (> 5 digits < 20 digits) will compress on average 50%.
- *string* values * of length <= 254 stay the same size. * of length > 254 are 3-bytes bigger per string.
- *object* and *array* values compress 1-byte-per-element.

One of the great things about the Universal Binary JSON format is that even though most all your data will be represented in a smaller footprint, you still get two big wins:

1. A smaller data format means faster writes and smaller reads. It also means less data to process when parsing.
2. Binary format means no encoding/decoding primitive values to text and no parsing primitive values from text.

1.6 Endianness

The Universal Binary JSON specification requires that all numeric values be written in Big-Endian order.

1.7 MIME Type

The Universal Binary JSON specification is a binary format and recommends using the following mime type:

```
application/x-ubjson
```

This was added directly to the specification in hopes of avoiding similar confusion with JSON.

1.8 File Extension

`ubj` is the recommended file extension when writing out files using the Universal Binary JSON format (e.g. `user.ubj`).

The extension stands for *"Universal Binary JSON"* and has no known conflicting mappings to other file formats.

1.9 Best Practices

Through work with the community, feedback from others and our own experience with the specification, below are some of the best-practices collected into one place making it easy for folks working with the format to find answers to the more flexible portions of the spec.

1.9.1 Handling *huge* Numbers

Not every language supports arbitrarily long numbers greater than 64-bits (represented by the *huge* data type), but many do.

If you are writing a library to read/write Universal Binary JSON and the platform you are working with does not support them, we recommend throwing an exception or returning an error to the caller, letting them know unsupported data is contained in the file they are trying to parse.

If the library you are writing is meant to be a general-purpose parser and needs to be more resilient than that, we recommend the following:

1. Make the default behavior to throw an exception or return an error when the unsupported huge data type is encountered.
2. Provide an optional behavior to the parser (that must be specifically enabled by the caller) that treats the huge value as a simple string and returns it to the caller to handle (e.g. insert in a database) if they need it.
3. Provide an optional behavior to the parser (again, that must be specifically enabled by the caller) to simply skip unsupported values.

This implementation should give the user the most functional experience working with your library and the Universal Binary JSON format while making it clear on their particular platform some data types could cause trouble; this is preferred to making the default operation to ignore the unsupported values.

CHAPTER 2

Type reference

The table below is a quick-reference for folks working closely with the Universal Binary JSON format that want all the information at their finger tips:

Type	Size	Marker	Length?	Data?
null	1-byte	Z	No	No
true	1-byte	T	No	No
false	1-byte	F	No	No
byte	2-bytes	B	No	Yes
int16	3-bytes	i	No	Yes
int32	5-bytes	I	No	Yes
int64	9-bytes	L	No	Yes
float (32-bit)	5-bytes	d	No	Yes
double (64-bit)	9-bytes	D	No	Yes
huge (number)	2-bytes + byte length of string	h	Yes	Yes if non-empty
huge (number)	5-bytes + byte length of string	H	Yes	Yes, if non-empty
string	2-bytes + byte length of string	s	Yes	Yes, if non-empty
string	5-bytes + byte length of string	S	Yes	Yes, if non-empty
array	2-bytes + byte length of string	a	Yes	Yes, if non-empty
array	5-bytes + byte length of string	A	Yes	Yes, if non-empty
object	2-bytes + byte length of string	o	Yes	Yes if non-empty
object	5-bytes + byte length of string	O	Yes	Yes, if non-empty
noop	1-byte	N	No	No
end	1-byte	E	No	No

2.1 Numeric Types

All numeric types are signed.

2.1.1 floats (32-bit)

All 32-bit float values are written into the binary format using the IEEE 754 single precision floating point format, which is the following structure:

- Bit 31 (1 bit) – sign
- Bit 30-23 (8 bits) – exponent
- Bit 22-0 (23 bits) – fraction (significand)

2.1.2 doubles (64-bit)

All 64-bit double values are written into the binary format using the IEEE 754 double precision floating point format, which is the following structure:

- Bit 63 (1 bit) – sign
- Bit 62-52 (11 bits) – exponent
- Bit 51-0 (52 bits) – fraction (significand)

2.2 String Encoding

All *string* values (which includes *huge* values since they are string-encoded) must be UTF-8 encoded.

This provides a number of advantages and inter-compatibility across systems and alternative data formats.

2.3 Arrays & Objects

The *length* argument specified is the *number of child elements* the parent container contains. A *child element* is defined as:

- in an *object*, a single name-value pair.
- in an *array*, a single value.

For example:

- if an array contains 4 integers, the *length* of that array is 4.
- if an object contains 4 name-value pairs, the *length* of that object is 4.
- if an array contains 13 *User objects*, the *length* of the array is 13.
- if an object contains 7 arrays, the *length* of the object is 7.

Note: Universal Binary JSON is a *streaming-friendly* specification and supports the use of *unknown-length container* types if you need them!

2.4 Support for 'huge' Numeric Type

The huge data type is an ultra-portable mechanism by which arbitrarily long numbers `> 64-bit` in size (integer or decimal) can be passed between systems that support them and degraded gracefully in systems that do not support them.

Note: *huge* values are **only** meant to store values `> 64-bit` in size. It is in violation of the Universal Binary JSON specification to store a value `<= 64-bits` as a huge.

This design was chosen intentionally as it greatly simplifies (and optimizes) the generation and parsing code for the UBJ format as no introspection of the *huge* value is necessary for a platform to try and marshal them into a smaller format.

This way the parsing code becomes simple, either creating an arbitrarily large number out of the value (e.g. BigDecimal in Java), returns an error to the caller because of an unsupported type or optionally skips the unsupported data and continues parsing.

huge values must be written out in accordance with the original JSON number specification.

Many programming languages have native support for arbitrarily large numbers, but many do not. If you are working in an environment that does not support numbers > 64-bit numbers, please see our recommendation on handling them in the *Best Practices* section.

2.5 Optimized Storage Size

All variable-length value types (*string*, *huge*, *array*, *object*) have a more compact representation using 1-byte (instead of 4-bytes) for the *length* argument when the *length* value is <= 254.

These more compact types always use the lowercased version of the *marker* ASCII char. For example, `a` for *array*, `s` for *string* and so on.

Warning: When using the compact representations of these different types, remember that the *length* must be <= `254` because the *length* of 255 (`0xFF`) has a special meaning when it comes to *array* and *object* types.

2.6 noop and Streaming Support

The *noop type* is a general purpose type that has no meaning, but is mostly commonly used in streaming scenarios where a server must send a client a *keep alive* message.

To support this use-case, the specification needed to support a special type that meant nothing, so a server and client could make use of it without polluting the actual data that was being exchanged.

Warning: The *noop* type can be used for other purposes or signals as well, but it is defined to have no value and no effect on the data it may be included in.
The *noop* type is meant to be sent between discrete values in a streaming scenario and can never be sent inside of the byte-data that makes up a single value.
For example, if a server is writing a string "Hello World" back to the client, the server must write the entire `[s][11][Hello World]` sequence back to the client unbroken; a *noop* marker cannot be sent inside of that value.
noop markers must only be written between values being transmitted (e.g. between values in an *array* or between the name and value pair inside of an *object*).

2.7 Examples

Please see the *Value Types* and *Container Types* sections of the specification for examples.

CHAPTER 3

Libraries

Below are a list of libraries, by language, that implement the Universal Binary JSON Specification.

3.1 D

- UBJSON for D

3.2 Java

- Universal Binary JSON Java Library

3.3 .NET

- Ubjson.NET

3.4 Node.js

- node-ubjson

3.5 Python

- simpleubjson

CHAPTER 4

Thanks

Below is a list of people that have submitted specific contributions, corrections and implementations to help make the Universal Binary JSON specification better.

Thank you all!

- Alex Blewitt

  Helped catch a number of specification errors around UTF-8 encoding in the original draft of the specification that would have been confusing/nasty to release. He also provided great feedback about the size and performance metrics for the specification.

- Alexander Shorin

  Alex is both the author of the Python library and a valued collaborator on the Universal Binary JSON spec as it matured. Alex provided instrumental insight into the modifications made between Draft 8 and Draft 9 of the spec to help simplify the spec by removing all the duplicate (compact) type representations, simplifying the length-arguments for *STRING* and *HUGE* as well as being the one to point out that the length arguments for the *ARRAY* and *OBJECT* container types are effectively useless once the streaming-format support was added (and do not make generator code or parsing code any easier or more performant).

- John Cowan

  John was the one that recommended using UTF-8 string-encoded values (or *huge*) for arbitrarily huge numbers after seeing my desire to avoid including any non-portable constructs into the binary format.

  Given that the discussion on numeric formats had been a very active one with lots of feelings on all sides, it was a boon to have John step up with such a simple suggestion that allowed for maximum compatibility and portability. It was a win-win all the way around.

- Michael Makarenko (aka "M1xA")

  Michael is the author behind the Ubjson.NET library and contributor of the *int16* and *float* numeric types to the specification. For numeric-heavy (e.g. scientific) data, the inclusions of the *int16* and *float* types can lead to significant space savings when writing out values in the Universal Binary JSON format.

  Michael has also gone to great lengths to make the .NET implementation of UBJSON as tight and performant as possible; collaborating on benchmark design and testing data as well as compatibility testing between implementations to ensure a great Universal Binary JSON experience for .NET developers.

  In addition to development, Michael has helped contribute to the growth of the Universal Binary JSON community with articles about the specification.

- Paul Davis

  While approaching the CouchDB team for feedback on the Universal Binary JSON spec, I met Paul who was willing to spend a significant amount of time reviewing the specification and recommending suggestions,

changes and improvements from everything the CouchDB team has learned by dealing closely with JSON for years.

Paul was the brains behind the compacted type presentation (`s`, `h`, `a` and `o`) using a single byte instead of 3 bytes to represent the length of an entity which was something the spec had originally avoided due to complexity, but as Paul clarified at-scale (e.g. a huge CouchDB data store) those few bytes in some data sets that are working with very small values (like string keywords) can really add up.

Paul also pointed out the shortcomings of prefixing the length to the two container types if the specification could ever be used easily with services or apps that streamed UBJ format for huge runs of data that the server couldn't load, buffer and count ahead of time before responding to the client. In order to more easily support streaming, unknown-length container types had to be added.

Paul also pointed out the importance of a `NO_OP`/`SKIP`/`IGNORE` type that can be useful during a long-lived streaming operation where the server may be waiting on something (like a DB) and you need to keep the connection alive between client/server and avoid the client timing out, but you need the client to know the data it is receiving is just meant as a "Hang on" message from the server and not actual data. This is where the `NO_OP` command comes in handy.

- Stephan Beal

  Stephan helped quite a bit with understanding the implications of a >= 64-bit numeric format and the implications of portability across a number of popular platforms.

- JSON Specification Group

  I would like to personally thank everyone in the JSON Specification Group. The amount of feedback and help with the specification has been wonderful, constructive and creative. It also lead to one of the busiest conversations in the last year!

CHAPTER 5

Why UBJSON?

Attempts to make using JSON faster through binary specifications like BSON, BJSON or Smile exist, but have been rejected from mass-adoption for two reasons:

- Custom (Binary-Only) Data Types: Inclusion of custom data types that have no ancillary in the original JSON spec, leaving room for incompatibilities to exist as different implementations of the spec handle the binary-only data types differently.
- Complexity: Some specifications provide higher performance or smaller representations at the cost of a much more complex specification, making implementations more difficult which can slow or block adoption. One of the key reasons JSON became as popular as it did was because of its ease of use.

BSON, for example, defines types for binary data, regular expressions, JavaScript code blocks and other constructs that have no equivalent data type in JSON. BJSON defines a binary data type as well, again leaving the door wide open to interpretation that can potentially lead to incompatibilities between two implementations of the spec and Smile, while the closest, defines more complex data constructs and generation/parsing rules in the name of absolute space efficiency.

The existing binary JSON specifications all define incompatibilities or complexities that undo the singular tenant that made JSON so successful: **simplicity**.

JSON's simplicity made it accessible to anyone, made implementations in every language available and made explaining it to anyone consuming your data immediate.

Any successful binary JSON specification must carry these properties forward for it to be genuinely helpful to the community at large.

This specification is defined around a singular construct used to build up and represent JSON values and objects. Reading and writing the format is trivial, designed with the goal of being understood in under 10 minutes (likely less if you are very comfortable with JSON already).

Fortunately, while the Universal Binary JSON specification carriers these tenants of simplicity forward, it is also able to take advantage of optimized binary data structures that are (on average) 30% smaller than compacted JSON and specified for ultimate read performance; bringing **simplicity**, **size** and **performance** all together into a single specification that is 100% compatible with JSON.

CHAPTER 6

Why not JSON+gzip?

On the surface simply gzipping your compacted JSON may seem like a valid (and smaller) alternative to using the Universal Binary JSON specification, but there are two significant costs associated with this approach that you should be aware of:

1. At least a 50% performance overhead for processing the data.
2. Lack of data clarity and inability to inspect it directly.

While gzipping your JSON will give you great compression, about 75% on average, the overhead required to read/write the data becomes significantly higher. Additionally, because the binary data is now in a compressed format you can no longer open it directly in an editor and scan the human-readable portions of it easily; which can be important during debugging, testing or data verification and recovery.

Utilizing the Universal Binary JSON format will typically provide a 30% reduction in size and store your data in a read-optimized format offering you much higher performance than even compacted JSON. If you had a usage scenario where your data is put into long-term cold storage and pulled out in large chunks for processing, you might even consider gzipping your Universal Binary JSON files, storing those, and when they are pulled out and unzipped, you can then process them with all the speed advantages of UBJ.

As always, deciding which approach is right for your project depends heavily on what you need.

CHAPTER 7

Goals

The Universal Binary JSON specification has 3 goals:

1. **Universal Compatibility**

   Meaning absolute compatibility with the JSON spec itself as well as only utilizing data types that are natively supported in all popular programming languages.

   This allows 1:1 transforms between standard JSON and Universal Binary JSON as well as efficient representation in all popular programming languages without requiring parser developers to account for strange data types that their language may not support.

2. **Ease of Use**

   The Universal Binary JSON specification is intentionally defined using a single core data structure to build up the entire specification.

   This accomplishes two things: it allows the spec to be understood quickly and allows developers to write trivially simple code to take advantage of it or interchange data with another system utilizing it.

3. **Speed / Efficiency**

   Typically the motivation for using a binary specification over a text-based one is speed and/or efficiency, so strict attention was paid to selecting data constructs and representations that are (roughly) 30% smaller than their compacted JSON counterparts and optimized for fast parsing.

CHAPTER 8

Indices and tables

- *genindex*
- *search*