

# JSON Lump formal specification version 1.0

A requirement to store data in JSON format has become needed. This document specifies how any JSON formatted data must be described when inserted into a Doom WAD file.

Rather than just be a JSON document containing the data relevant to a given feature, it is expected for verification and versioning purposes that any lump containing JSON data will provide some common information that both the engine and toolsets can support. A standard header is thus defined that any Doom source port can read and interpret to make base-level decisions on.

## JSON ISO specification

JSON lumps are expected to support the ISO/IEC 21778:2017 standard for JSON data. It is expected to be strictly adhered to. Any fields that fall outside of the scope of that standard (including comments) are to be considered an error and result in an invalid JSON document.

## Identifying JSON lumps

What is and isn't a JSON lump at a name level is outside of the scope of this specification. It is entirely up to the specifications for JSON-based lumps to define naming conventions for those lumps.

Once you have the lump data in memory, a JSON lump must be identified solely by attempting to parse it as a JSON document. Your JSON implementation must immediately return a fail if it is not a correctly-formatted JSON document.

For lumps that have multiple existing types (ie binary formats, text, etc), JSON is to be considered the first class citizen and must attempt to be parsed first.

Note that attempting to shortcut this requirement by searching for the first non-whitespace character and matching it to '{' is not supported and is not to be implemented. Many binary formats used by the Doom engine start off with little-endian values that can hit the binary value of 123 that the '{' character is encoded at in both ASCII and UTF-8 standards. Either attempt to parse a full JSON document, or only use the original handlers.

## JSON structure

The root node of a JSON lump is required to have **all of- and only-** the values described below. Any values encountered not specified in the below table are to be considered an error condition and result in an invalid JSON document.

All values have a default of null. Any values not loaded thus resolve to null, and are to be considered an error condition and result in an invalid JSON document.

Any values that do not conform to specified types are to be considered an error condition and result in an invalid JSON document.

Any JSON data handler encountering a version higher than expected must consider it to be an error condition and result in an invalid JSON document. Forward compatibility is not supported through this requirement, only backward compatibility.

Name	JSON type	Description
<b>type</b>	string	<p>The name of the type of data this lump contains. Used for verification purposes at engine runtime; used by editors to determine how to handle the lump.</p> <p>This value must be lowercase; have one character minimum; and consist of letters, numbers, underscores, and dashes only. The regex string <code>^[a-z0-9_-]+\$</code> should thus result in an exact match with the provided value.</p>
<b>version</b>	string	<p>The version for the specified type. Expected to take the format of “&lt;major&gt;.&lt;minor&gt;.&lt;revision&gt;”. The regex string <code>^(\d+)\.(\d+)\.(\d+)\$</code> should thus result in an exact match and three groups with the provided value.</p>
<b>metadata</b>	object	<p>A data field designed to be used by editors. Its existence must be verified at engine runtime; and its contents must be ignored at engine runtime.</p>
<b>data</b>	object	<p>The data representing the specified type. The format of this block is out of the scope of this document, and intended to be defined exclusively by any data formats wanting to base themselves on JSON. This object is thus used as the root node for any custom JSON data handler, any handlers as such must refer to the type of this object as <b>root</b>. Must not be null.</p>

### metadata structure

While the metadata object must be ignored at runtime, the contents of the metadata object must contain a few common values. Any tool parsing this block can thus rely on these fields existing, and not need to deal with special-case handlers for the way other tools wish to work.

Each field has no default value. Null is an acceptable default value for verification purposes.

Any metadata block not containing all of the following entries must be considered invalid

Any additional values not defined here are both valid and outside of the scope of this document, and are to be considered “implementation defined” for the purposes of this document.

Name	JSON type	Example	Description
<b>author</b>	string	Ethan Watson	The name(s) of the author(s) of this lump.
<b>timestamp</b>	string	1993-12-10T00:30:00-06:00	The timestamp representing when this lump was last modified, formatted to RFC3339 specifications. The example is a timestamp representing roughly when the Doom shareware episode was first available for the public to download.
<b>application</b>	string	DEU 5.21	The name of the application used to create this lump. While it is strongly recommended to use an application and not hand-editing JSON, if you insist on hand-editing things then by all means brag about it here.

## Versioning

The **revision** field of a version is meant primarily for bug fixes and alterations to a published standard. Any additions or removals of features should be saved for a **minor** version; however, it is up to the author's discretion as to whether that is a big enough functionality change to be considered a **major** version change.

## Requirements of specifications based on the JSON Lump specification

Preferably at the top of your specification, you must define the following:

- JSON Lump version specification that your specification is based on
- The value that the **type** field must be set to
- The value that the **version** field must be set to
- A definition for a **root** object that is stored in the **data** field

## Reference implementation details

### API implementation

The entrypoint to the JSON handling API looks like:

```
jsonlumpresult_t M_ParseJSONLump( lumpindex_t lumpindex, const char*  
lumptype, const JSONLumpVersion& maxversion, const JSONLumpFunc& parsefunc  
);
```

Upon successfully validating the above specification, `M_ParseJSONLump` will then invoke `parsefunc` with the node representing the data field. In general, it is invoked with a lambda that captures a reference to an output value. This allows `M_ParseJSONLump` to provide error reporting as a return value without resorting to exceptions. The closure created to capture the reference is also quite small, and is effectively no more costly than invoking a virtual method on a class.

#### JSONLumpVersion struct

The `JSONLumpVersion` struct contains numerical representations of the **major**, **minor**, and **revision** fields. It also contains a **devversion** field. This is intended to be used exclusively during development of a new revision of a feature. As it is not serialised anywhere or specified in the standard, this value should never be relied on for a published specification or data that relies on any given version of a specification.