Jansson Documentation

Release 1.1.3

Petri Lehtinen

August 20, 2013

CONTENTS

This is the documentation for Jansson 1.1.3, last updated August 20, 2013.

INTRODUCTION

Jansson is a C library for encoding, decoding and manipulating JSON data. Its main features and design principles are:

- Simple and intuitive API and data model

- Comprehensive documentation

- No dependencies on other libraries

- Full Unicode support (UTF-8)

- Extensive test suite

Jansson is licensed under the MIT license; see LICENSE in the source distribution for details.

CHAPTER

TWO

CONTENTS

2.1 Getting Started

2.1.1 Compiling and Installing Jansson

The Jansson source is available at http://www.digip.org/jansson/releases/.

Unpack the source tarball and change to the source directory:

```
bunzip2 -c jansson-1.1.3.tar.bz2 | tar xf -
cd jansson-1.1.3
```

The source uses GNU Autotools (autoconf, automake, libtool), so compiling and installing is extremely simple:

```
./configure
make
make check
make install
```

To change the destination directory (`/usr/local` by default), use the `--prefix=DIR` argument to `./configure`. See `./configure --help` for the list of all possible installation options. (There are no options to customize the resulting Jansson binary.)

The command `make check` runs the test suite distributed with Jansson. Python is required to run the tests. This step is not strictly necessary, but it may find possible problems that Jansson has on your platform. If any problems are found, please report them.

If you obtained the source from a Git repository (or any other source control system), there's no `./configure` script as it's not kept in version control. To create the script, Autotools needs to be bootstrapped. There are many ways to do this, but the easiest one is to use `autoreconf`:

```
autoreconf -vi
```

This command creates the `./configure` script, which can then be used as described in the previous section.

Installing Prebuilt Binary Packages

Binary `.deb` packages for Ubuntu are available in this PPA at Launchpad. Follow the instructions in the PPA ("Technical details about this PPA" link) to take the PPA into use. Then install the -dev package:

```
sudo apt-get install libjansson-dev
```

5

Building the Documentation

(This subsection describes how to build the HTML documentation you are currently reading, so it can be safely skipped.)

Documentation is in the `doc/` subdirectory. It's written in reStructuredText with Sphinx annotations. To generate the HTML documentation, invoke:

```
cd doc/
sphinx-build . .build/html
```

... and point your browser to `.build/html/index.html`. Sphinx is required to generate the documentation.

2.1.2 Compiling Programs Using Jansson

Jansson involves one C header file, `jansson.h`, so it's enough to put the line

```
#include <jansson.h>
```

in the beginning of every source file that uses Jansson.

There's also just one library to link with, `libjansson`. Compile and link the program as follows:

```
cc -o prog prog.c -ljansson
```

2.2 Tutorial

In this tutorial, we create a program that fetches the latest commits of a repository in GitHub over the web. One of the response formats supported by GitHub API is JSON, so the result can be parsed using Jansson.

To stick to the the scope of this tutorial, we will only cover the the parts of the program related to handling JSON data. For the best user experience, the full source code is available: `github_commits.c`. To compile it (on Unix-like systems with gcc), use the following command:

```
gcc -o github_commits github_commits.c -ljansson -lcurl
```

libcurl is used to communicate over the web, so it is required to compile the program.

The command line syntax is:

```
github_commits USER REPOSITORY
```

`USER` is a GitHub user ID and `REPOSITORY` is the repository name. Please note that the GitHub API is rate limited, so if you run the program too many times within a short period of time, the sever starts to respond with an error.

2.2.1 The GitHub Commits API

The GitHub commits API is used by sending HTTP requests to URLs starting with `http://github.com/api/v2/json/commits/`. Our program only lists the latest commits, so the rest of the URL is `list/USER/REPOSITORY/BRANCH`, where `USER`, `REPOSITORY` and `BRANCH` are the GitHub user ID, the name of the repository, and the name of the branch whose commits are to be listed, respectively.

GitHub responds with a JSON object of the following form:

```
{
    "commits": [
        {
            "id": "<the commit ID>",
            "message": "<the commit message>",
            <more fields, not important to this tutorial>
        },
        {
            "id": "<the commit ID>",
            "message": "<the commit message>",
            <more fields, not important to this tutorial>
        },
        <more commits...>
    ]
}
```

In our program, the HTTP request is sent using the following function:

```c
static char *request(const char *url);
```

It takes the URL as a parameter, preforms a HTTP GET request, and returns a newly allocated string that contains the response body. If the request fails, an error message is printed to stderr and the return value is *NULL*. For full details, refer to `the code`, as the actual implementation is not important here.

2.2.2 The Program

First the includes:

```c
#include <string.h>
#include <jansson.h>
```

Like all the programs using Jansson, we need to include `jansson.h`.

The following definitions are used to build the GitHub commits API request URL:

```c
#define URL_FORMAT   "http://github.com/api/v2/json/commits/list/%s/%s/master"
#define URL_SIZE     256
```

The following function is used when formatting the result to find the first newline in the commit message:

```c
/* Return the offset of the first newline in text or the length of
   text if there's no newline */
static int newline_offset(const char *text)
{
    const char *newline = strchr(text, '\n');
    if(!newline)
        return strlen(text);
    else
        return (int)(newline - text);
}
```

The main function follows. In the beginning, we first declare a bunch of variables and check the command line parameters:

```c
unsigned int i;
char *text;
char url[URL_SIZE];
```

```
json_t *root;
json_error_t error;
json_t *commits;

if(argc != 3)
{
    fprintf(stderr, "usage: %s USER REPOSITORY\n\n", argv[0]);
    fprintf(stderr, "List commits at USER's REPOSITORY.\n\n");
    return 2;
}
```

Then we build the request URL using the user and repository names given as command line parameters:

```
snprintf(url, URL_SIZE, URL_FORMAT, argv[1], argv[2]);
```

This uses the URL_SIZE and URL_FORMAT constants defined above. Now we're ready to actually request the JSON data over the web:

```
text = request(url);
if(!text)
    return 1;
```

If an error occurs, our function request prints the error and returns *NULL*, so it's enough to just return 1 from the main function.

Next we'll call json_loads() to decode the JSON text we got as a response:

```
root = json_loads(text, &error);
free(text);

if(!root)
{
    fprintf(stderr, "error: on line %d: %s\n", error.line, error.text);
    return 1;
}
```

We don't need the JSON text anymore, so we can free the text variable right after decoding it. If json_loads() fails, it returns *NULL* and sets error information to the json_error_t structure given as the second parameter. In this case, our program prints the error information out and returns 1 from the main function.

Now we're ready to extract the data out of the decoded JSON response. The structure of the response JSON was explained in section *The GitHub Commits API*.

First, we'll extract the commits array from the JSON response:

```
commits = json_object_get(root, "commits");
if(!json_is_array(commits))
{
    fprintf(stderr, "error: commits is not an array\n");
    return 1;
}
```

This is the array that contains objects describing latest commits in the repository. We check that the returned value really is an array. If the key commits doesn't exist, json_object_get() returns *NULL*, but json_is_array() handles this case, too.

Then we proceed to loop over all the commits in the array:

```
for(i = 0; i < json_array_size(commits); i++)
{
    json_t *commit, *id, *message;
```

```
    const char *message_text;

    commit = json_array_get(commits, i);
    if(!json_is_object(commit))
    {
        fprintf(stderr, "error: commit %d is not an object\n", i + 1);
        return 1;
    }
...
```

The function `json_array_size()` returns the size of a JSON array. First, we again declare some variables and then extract the i'th element of the `commits` array using `json_array_get()`. We also check that the resulting value is a JSON object.

Next we'll extract the commit ID and commit message, and check that they both are JSON strings:

```
    id = json_object_get(commit, "id");
    if(!json_is_string(id))
    {
        fprintf(stderr, "error: commit %d: id is not a string\n", i + 1);
        return 1;
    }

    message = json_object_get(commit, "message");
    if(!json_is_string(message))
    {
        fprintf(stderr, "error: commit %d: message is not a string\n", i + 1);
        return 1;
    }
...
```

And finally, we'll print the first 8 characters of the commit ID and the first line of the commit message. A C-style string is extracted from a JSON string using `json_string_value()`:

```
    message_text = json_string_value(message);
    printf("%.8s %.*s\n",
            json_string_value(id),
            newline_offset(message_text),
            message_text);
}
```

After sending the HTTP request, we decoded the JSON text using `json_loads()`, remember? It returns a *new reference* to the JSON value it decodes. When we're finished with the value, we'll need to decrease the reference count using `json_decref()`. This way Jansson can release the resources:

```
json_decref(root);
return 0;
```

For a detailed explanation of reference counting in Jansson, see *Reference Count* in *API Reference*.

The program's ready, let's test it and view the latest commits in Jansson's repository:

```
$ ./github_commits akheron jansson
86dc1d62 Fix indentation
b67e130f json_dumpf: Document the output shortage on error
4cd77771 Enhance handling of circular references
79009e62 json_dumps: Close the strbuffer if dumping fails
76999799 doc: Fix a small typo in apiref
22af193a doc/Makefile.am: Remove *.pyc in clean
951d091f Make integer, real and string mutable
```

```
185e107d Don't use non-portable asprintf()
ca7703fb Merge branch '1.0'
12cd4e8c jansson 1.0.4
<etc...>
```

2.2.3 Conclusion

In this tutorial, we implemented a program that fetches the latest commits of a GitHub repository using the GitHub commits API. Jansson was used to decode the JSON response and to extract the commit data.

This tutorial only covered a small part of Jansson. For example, we did not create or manipulate JSON values at all. Proceed to *API Reference* to explore all features of Jansson.

2.3 API Reference

2.3.1 Preliminaries

All declarations are in `jansson.h`, so it's enough to

```
#include <jansson.h>
```

in each source file.

All constants are prefixed `JSON_` and other identifiers with `json_`. Type names are suffixed with `_t` and `typedef`'d so that the `struct` keyword need not be used.

2.3.2 Value Representation

The JSON specification (RFC 4627) defines the following data types: *object*, *array*, *string*, *number*, *boolean*, and *null*. JSON types are used dynamically; arrays and objects can hold any other data type, including themselves. For this reason, Jansson's type system is also dynamic in nature. There's one C type to represent all JSON values, and this structure knows the type of the JSON value it holds.

json_t
    This data structure is used throughout the library to represent all JSON values. It always contains the type of the JSON value it holds and the value's reference count. The rest depends on the type of the value.

Objects of `json_t` are always used through a pointer. There are APIs for querying the type, manipulating the reference count, and for constructing and manipulating values of different types.

Unless noted otherwise, all API functions return an error value if an error occurs. Depending on the function's signature, the error value is either *NULL* or -1. Invalid arguments or invalid input are apparent sources for errors. Memory allocation and I/O operations may also cause errors.

Type

The type of a JSON value is queried and tested using the following functions:

enum **json_type**
    The type of a JSON value. The following members are defined:

JSON_OBJECT
JSON_ARRAY
JSON_STRING
JSON_INTEGER
JSON_REAL
JSON_TRUE
JSON_FALSE
JSON_NULL

These correspond to JSON object, array, string, number, boolean and null. A number is represented by either a value of the type JSON_INTEGER or of the type JSON_REAL. A true boolean value is represented by a value of the type JSON_TRUE and false by a value of the type JSON_FALSE.

int **json_typeof** (const *json_t *json*)

Return the type of the JSON value (a json_type cast to int). *json* MUST NOT be *NULL*. This function is actually implemented as a macro for speed.

json_is_object (const *json_t *json*)
json_is_array (const *json_t *json*)
json_is_string (const *json_t *json*)
json_is_integer (const *json_t *json*)
json_is_real (const *json_t *json*)
json_is_true (const *json_t *json*)
json_is_false (const *json_t *json*)
json_is_null (const *json_t *json*)

These functions (actually macros) return true (non-zero) for values of the given type, and false (zero) for values of other types and for *NULL*.

json_is_number (const *json_t *json*)

Returns true for values of types JSON_INTEGER and JSON_REAL, and false for other types and for *NULL*.

json_is_boolean (const *json_t *json*)

Returns true for types JSON_TRUE and JSON_FALSE, and false for values of other types and for *NULL*.

Reference Count

The reference count is used to track whether a value is still in use or not. When a value is created, it's reference count is set to 1. If a reference to a value is kept (e.g. a value is stored somewhere for later use), its reference count is incremented, and when the value is no longer needed, the reference count is decremented. When the reference count drops to zero, there are no references left, and the value can be destroyed.

The following functions are used to manipulate the reference count.

json_t ***json_incref** (*json_t *json*)

Increment the reference count of *json* if it's not non-*NULL*. Returns *json*.

void **json_decref** (*json_t *json*)

Decrement the reference count of *json*. As soon as a call to json_decref() drops the reference count to zero, the value is destroyed and it can no longer be used.

Functions creating new JSON values set the reference count to 1. These functions are said to return a **new reference**. Other functions returning (existing) JSON values do not normally increase the reference count. These functions are said to return a **borrowed reference**. So, if the user will hold a reference to a value returned as a borrowed reference, he must call json_incref(). As soon as the value is no longer needed, json_decref() should be called to release the reference.

Normally, all functions accepting a JSON value as an argument will manage the reference, i.e. increase and decrease the reference count as needed. However, some functions **steal** the reference, i.e. they have the same result as if the user

called `json_decref()` on the argument right after calling the function. These are usually convenience functions for adding new references to containers and not to worry about the reference count.

In the following sections it is clearly documented whether a function will return a new or borrowed reference or steal a reference to its argument.

Circular References

A circular reference is created when an object or an array is, directly or indirectly, inserted inside itself. The direct case is simple:

```
json_t *obj = json_object();
json_object_set(obj, "foo", obj);
```

Jansson will refuse to do this, and `json_object_set()` (and all the other such functions for objects and arrays) will return with an error status. The indirect case is the dangerous one:

```
json_t *arr1 = json_array(), *arr2 = json_array();
json_array_append(arr1, arr2);
json_array_append(arr2, arr1);
```

In this example, the array `arr2` is contained in the array `arr1`, and vice versa. Jansson cannot check for this kind of indirect circular references without a performance hit, so it's up to the user to avoid them.

If a circular reference is created, the memory consumed by the values cannot be freed by `json_decref()`. The reference counts never drops to zero because the values are keeping the circular reference to themselves. Moreover, trying to encode the values with any of the encoding functions will fail. The encoder detects circular references and returns an error status.

2.3.3 True, False and Null

These values are implemented as singletons, so each of these functions returns the same value each time.

json_t *__json_true__ (void)
> *Return value: New reference.* Returns the JSON true value.

json_t *__json_false__ (void)
> *Return value: New reference.* Returns the JSON false value.

json_t *__json_null__ (void)
> *Return value: New reference.* Returns the JSON null value.

2.3.4 String

json_t *__json_string__ (const char *value)
> *Return value: New reference.* Returns a new JSON string, or *NULL* on error. *value* must be a valid UTF-8 encoded Unicode string.

const char *__json_string_value__ (const json_t *string)
> Returns the associated value of *string* as a null terminated UTF-8 encoded string, or *NULL* if *string* is not a JSON string.

int __json_string_set__ (const json_t *string, const char *value)
> Sets the associated value of *string* to *value*. *value* must be a valid UTF-8 encoded Unicode string. Returns 0 on success and -1 on error. New in version 1.1.

2.3.5 Number

json_t ***json_integer** (int *value*)
> *Return value: New reference.* Returns a new JSON integer, or *NULL* on error.

int **json_integer_value** (const json_t *integer*)
> Returns the associated value of *integer*, or 0 if *json* is not a JSON integer.

int **json_integer_set** (const json_t *integer*, int *value*)
> Sets the associated value of *integer* to *value*. Returns 0 on success and -1 if *integer* is not a JSON integer. New in version 1.1.

json_t ***json_real** (double *value*)
> *Return value: New reference.* Returns a new JSON real, or *NULL* on error.

double **json_real_value** (const json_t *real*)
> Returns the associated value of *real*, or 0.0 if *real* is not a JSON real.

int **json_real_set** (const json_t *real*, double *value*)
> Sets the associated value of *real* to *value*. Returns 0 on success and -1 if *real* is not a JSON real. New in version 1.1.

In addition to the functions above, there's a common query function for integers and reals:

double **json_number_value** (const json_t *json*)
> Returns the associated value of the JSON integer or JSON real *json*, cast to double regardless of the actual type. If *json* is neither JSON real nor JSON integer, 0.0 is returned.

2.3.6 Array

A JSON array is an ordered collection of other JSON values.

json_t ***json_array** (void)
> *Return value: New reference.* Returns a new JSON array, or *NULL* on error. Initially, the array is empty.

unsigned int **json_array_size** (const json_t *array*)
> Returns the number of elements in *array*, or 0 if *array* is NULL or not a JSON array.

json_t ***json_array_get** (const json_t *array*, unsigned int *index*)
> *Return value: Borrowed reference.* Returns the element in *array* at position *index*. The valid range for *index* is from 0 to the return value of `json_array_size()` minus 1. If *array* is not a JSON array, if *array* is *NULL*, or if *index* is out of range, *NULL* is returned.

int **json_array_set** (json_t *array*, unsigned int *index*, json_t *value*)
> Replaces the element in *array* at position *index* with *value*. The valid range for *index* is from 0 to the return value of `json_array_size()` minus 1. Returns 0 on success and -1 on error.

int **json_array_set_new** (json_t *array*, unsigned int *index*, json_t *value*)
> Like `json_array_set()` but steals the reference to *value*. This is useful when *value* is newly created and not used after the call. New in version 1.1.

int **json_array_append** (json_t *array*, json_t *value*)
> Appends *value* to the end of *array*, growing the size of *array* by 1. Returns 0 on success and -1 on error.

int **json_array_append_new** (json_t *array*, json_t *value*)
> Like `json_array_append()` but steals the reference to *value*. This is useful when *value* is newly created and not used after the call. New in version 1.1.

int **json_array_insert** (json_t *array*, unsigned int *index*, json_t *value*)
> Inserts *value* to *array* at position *index*, shifting the elements at *index* and after it one position towards the end of the array. Returns 0 on success and -1 on error. New in version 1.1.

int **json_array_insert_new** (json_t *array, unsigned int *index*, json_t *value*)
> Like json_array_insert() but steals the reference to *value*. This is useful when *value* is newly created and not used after the call. New in version 1.1.

int **json_array_remove** (json_t *array, unsigned int *index*)
> Removes the element in *array* at position *index*, shifting the elements after *index* one position towards the start of the array. Returns 0 on success and -1 on error. New in version 1.1.

int **json_array_clear** (json_t *array)
> Removes all elements from *array*. Returns 0 on sucess and -1 on error. New in version 1.1.

int **json_array_extend** (json_t *array, json_t *other_array*)
> Appends all elements in *other_array* to the end of *array*. Returns 0 on success and -1 on error. New in version 1.1.

2.3.7 Object

A JSON object is a dictionary of key-value pairs, where the key is a Unicode string and the value is any JSON value.

json_t *****json_object** (void)
> *Return value: New reference.* Returns a new JSON object, or *NULL* on error. Initially, the object is empty.

unsigned int **json_object_size** (const json_t *object*)
> Returns the number of elements in *object*, or 0 if *object* is not a JSON object. New in version 1.1.

json_t *****json_object_get** (const json_t *object, const char *key*)
> *Return value: Borrowed reference.* Get a value corresponding to *key* from *object*. Returns *NULL* if *key* is not found and on error.

int **json_object_set** (json_t *object, const char *key, json_t *value*)
> Set the value of *key* to *value* in *object*. *key* must be a valid null terminated UTF-8 encoded Unicode string. If there already is a value for *key*, it is replaced by the new value. Returns 0 on success and -1 on error.

int **json_object_set_new** (json_t *object, const char *key, json_t *value*)
> Like json_object_set() but steals the reference to *value*. This is useful when *value* is newly created and not used after the call. New in version 1.1.

int **json_object_del** (json_t *object, const char *key*)
> Delete *key* from *object* if it exists. Returns 0 on success, or -1 if *key* was not found.

int **json_object_clear** (json_t *object)
> Remove all elements from *object*. Returns 0 on success and -1 if *object* is not a JSON object. New in version 1.1.

int **json_object_update** (json_t *object, json_t *other*)
> Update *object* with the key-value pairs from *other*, overwriting existing keys. Returns 0 on success or -1 on error. New in version 1.1.

The following functions implement an iteration protocol for objects:

void *****json_object_iter** (json_t *object)
> Returns an opaque iterator which can be used to iterate over all key-value pairs in *object*, or *NULL* if *object* is empty.

void *****json_object_iter_next** (json_t *object, void *iter*)
> Returns an iterator pointing to the next key-value pair in *object* after *iter*, or *NULL* if the whole object has been iterated through.

const char *****json_object_iter_key** (void *iter*)
> Extract the associated key from *iter*.

json_t ***json_object_iter_value** (void *iter)
>   *Return value: Borrowed reference.* Extract the associated value from *iter*.

The iteration protocol can be used for example as follows:

```c
/* obj is a JSON object */
const char *key;
json_t *value;
void *iter = json_object_iter(obj);
while(iter)
{
    key = json_object_iter_key(iter);
    value = json_object_iter_value(iter);
    /* use key and value ... */
    iter = json_object_iter_next(obj, iter);
}
```

2.3.8 Encoding

This sections describes the functions that can be used to encode values to JSON. Only objects and arrays can be encoded, since they are the only valid "root" values of a JSON text.

Each function takes a *flags* parameter that controls some aspects of how the data is encoded. Its default value is 0. The following macros can be ORed together to obtain *flags*.

JSON_INDENT(n) Pretty-print the result, indenting arrays and objects by *n* spaces. The valid range for *n* is between 0 and 255, other values result in an undefined output. If JSON_INDENT is not used or *n* is 0, no pretty-printing is done and the result is a compact representation.

The following functions perform the actual JSON encoding. The result is in UTF-8.

char ***json_dumps** (const json_t *root*, unsigned long *flags*)
>   Returns the JSON representation of *root* as a string, or *NULL* on error. *flags* is described above. The return value must be freed by the caller using free().

int **json_dumpf** (const json_t *root*, FILE *output*, unsigned long *flags*)
>   Write the JSON representation of *root* to the stream *output*. *flags* is described above. Returns 0 on success and -1 on error. If an error occurs, something may have already been written to *output*. In this case, the output is undefined and most likely not valid JSON.

int **json_dump_file** (const json_t *json*, const char *path*, unsigned long *flags*)
>   Write the JSON representation of *root* to the file *path*. If *path* already exists, it is overwritten. *flags* is described above. Returns 0 on success and -1 on error.

2.3.9 Decoding

This sections describes the functions that can be used to decode JSON text to the Jansson representation of JSON data. The JSON specification requires that a JSON text is either a serialized array or object, and this requirement is also enforced with the following functions.

The only supported character encoding is UTF-8 (which ASCII is a subset of).

json_error_t
>   This data structure is used to return information on decoding errors from the decoding functions. Its definition is repeated here:

```
#define JSON_ERROR_TEXT_LENGTH  160

typedef struct {
    char text[JSON_ERROR_TEXT_LENGTH];
    int line;
} json_error_t;
```

line is the line number on which the error occurred, or -1 if this information is not available. *text* contains the error message (in UTF-8), or an empty string if a message is not available.

The normal usef of json_error_t is to allocate it normally on the stack, and pass a pointer to a decoding function. Example:

```
int main() {
    json_t *json;
    json_error_t error;

    json = json_load_file("/path/to/file.json", &error);
    if(!json) {
        /* the error variable contains error information */
    }
    ...
}
```

Also note that if the decoding succeeded (json != NULL in the above example), the contents of error are unspecified.

All decoding functions also accept *NULL* as the json_error_t pointer, in which case no error information is returned to the caller.

The following functions perform the actual JSON decoding.

json_t ***json_loads** (const char *input, json_error_t *error)

> *Return value: New reference.* Decodes the JSON string *input* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. See above for discussion on the *error* parameter.

json_t ***json_loadf** (FILE *input, json_error_t *error)

> *Return value: New reference.* Decodes the JSON text in stream *input* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. See above for discussion on the *error* parameter.

json_t ***json_load_file** (const char *path, json_error_t *error)

> *Return value: New reference.* Decodes the JSON text in file *path* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. See above for discussion on the *error* parameter.

INDICES AND TABLES

- *genindex*
- *search*