tangentsky Documentation

Josh Bialkowski

Mar 19, 2020

Contents:

A C++ library for working with JavaScript Object Notation.

The design of this library makes it especialy suitable for use in embedded applications. In particular, the design supports:

1. Parse into static structures with no memory allocations
2. Serialize into fixed output buffers with no memory allocations
3. No dependency on the standard library

Warning: While the design of the library enables these features, the current implementation doesn't quite meet these goals. For the most part there's a few standard library uses that need to be replaced with fixed-memory data structures and some of the standard library support is written in the same files as the core library. These will be segregated into their own files soon.

CHAPTER 1

Low Level API

There is a straight-forward, low level API that you may find suitable for working with JSON documents. It follows a typical lexer / parser pattern. You can manage the lex and parse steps separately with `json::Scanner` and `json::Parser`, or you can you use the thin combination `json::LexerParser`.

To work with the parse stream from a text document:

```
int DoParseStream(const std::string& content, std::ostream* log) {
  json::LexerScanner stream{};
  json::Error{};

  int status = stream.Init(&error);
  if(status < 0){
    (*log) << json::Error::ToString(error.code) << ": " << error.msg;
    return status;
  }

  stream.Begin(content);

  json::Event event{}
  while(stream.GetNextEvent(&event, &error) == 0){
    switch(event.typeno){
      // Emitted on the start of a json object (i.e. the '{' token)
      case json::Event::OBJECT_BEGIN:

      // Emitted when a key is parsed. The event.token.spelling contains the
      // content of the key as a string literal (i.e. including the double
      // quotes)
      case json::Event::OBJECT_KEY:

      // Emitted on the end of a json object (i.e. the '}' token)
      case json::Event::OBJECT_END:

      // Emitted on the start of a json list (i.e. the '[' token)
      case json::Event::LIST_BEGIN:
```

(continues on next page)

(continued from previous page)

```
      // Emitted on the end of a json list (i.e. the ']' token)
      case json::Event::LIST_END:

      // Emitted on any value literal including numeric, string, null, or
      // boolean.
      case json::Event::VALUE_LITERAL:
        break;
    }
  }

  if(error.code != json::Error::LEX_INPUT_FINISHED){
    (*log) << json::Error::ToString(error.code) << ": " << error.msg;
    return -1;
  } else {
    return 0;
  }
}
```

To work with a token and event stream directly from a text document:

```
int DoTokenStream(const std::string& content, std::ostream* log) {
  json::Scanner scanner{};
  json::Parser  parser{};
  json::Error{};

  int status = scanner.Init(&error);
  if(status < 0){
    (*log) << json::Error::ToString(error.code) << ": " << error.msg;
    return -1;
  }

  scanner.Begin(content);

  json::Token token{};
  json::Event event{};
  while(scanner.Pump(&token, &error) == 0){
    int status = parser.HandleToken(token, &event, &error);
    // error
    if(status < 0){
      (*log) << json::Error::ToString(error.code) << ": " << error.msg;
      return -1;
    } else if(status > 0){
      // An actionable event has occured, do something with the event
    } else {
      // No actionable event, but you can do something with the token
      // if you want. This means the token is either whitespace, colon,
      // or comma.
    }
  }

  if(error.code != json::Error::LEX_INPUT_FINISHED){
    return -1;
  } else {
    return 0;
  }
}
```

CHAPTER 2

Stream API

The "stream" API is meant to provide a mechanism for creating high level bindings for parsing a JSON character buffer directly into static objects, and for dumping statically defined structures out to JSON. You can use it to make your C/C++ structures JSON-serializable.

2.1 Parsing

The JSON event stream is provided by the `LexerParser` which maintains a state machine for the tokenization and semantic parsing of a JSON text document. It provides the `GetNextEvent()` which sequentially returns each semantic event in the stream.

Parsing is managed by a `stream::Registry` object, which acts as a kind of plugin registry storing pointers to parse functions. There are two types of parsers that are stored in the registry:

1. Field parsers: which basically just encode a list of fields for a particular type and can dispatch a typesafe parse for each.
2. Scalar parsers: which know how to turn a string into value of some particular type.

The library includes Scalar parsers for all the built-in numeric types and some string types. The field parsers are usually what you want to add. A field parser for a type `T` is a function with the signature:

```
int parse_T(
    const stream::Registry& registry, const re2::StringPiece& fieldname,
    LexerParser* event_stream, T* value);
```

The implementation of a field parser is pretty standard, just providing a dispatch for each field name and usually lookes this:

```
int parse_T(
    const json::stream::Registry& registry,
    const re2::StringPiece& fieldname,
    json::LexerParser* event_stream, T* value){
  int fieldno = json::RuntimeHash(fieldname);
```

(continues on next page)

(continued from previous page)

```
  switch(fieldno){
    case json::Hash("foo"):
      return registry.parse_value(event_stream, &value->foo);
    case json::Hash("bar"):
      return registry.parse_value(event_stream, &value->bar);
    default:
      json::sink_value(event_stream);
      break;
  }
  return 1;
}
```

In fact the above function definition can be generated automatically using the macro `JSON_DEFN(T, foo, bar);` , or by utilizing the `json_gen` code generator.

For scalar types, the signature is:

```
void parse_T(const json::Token& token, T* value);
```

Scalar parsers taking in a `json::Token` generally look at the *type* and *spelling* fields to parse the token into a scalar value. Most of the json native types map easily to native C++ types and a *Registry* already includes scalar parsers for these types (so you don't have to write them).

2.2 Dumping

Outputting JSON is done through a *stream::Dumper* object. A *Dumper* knows how to write out all of the native C++ numeric types and several string types. For object types you must provide a field-dumper, which is a function with the signature:

```
int dumpfields_T(const T& value, Dumper* dumper);
```

The implementation, like *parsefields* is usually pretty standard and looks something like this:

```
int dumpfields_T(const T& value, Dumper* dumper){
   int result = 0;
   result |= dumper->dump_field("foo", value->foo);
   result |= dumper->dump_field("bar", value->bar);
   result |= dumper->dump_field("baz", value->baz);
   return result;
}
```

In fact the above function definition can be generated automatically using the macro `JSON_DEFN(T, foo, bar);` , or by utilizing the `json_gen` code generator.

For scalar types, the signature is the same, but the implementation usually looks something like this:

```
int dumpscalar_U(const U& value, Dumper* dumper){
  dumper->dump_primitive(static_cast<double>(value));
}
```

The *Dumper* interface implements *dump_primitive* for all of the built-in c++ native numerical types and several common string types. For custom types, you will need to implement a conversion to one of these types.

2.3 The Global Registry

There is a global registry provided for convenience available through *json::stream::global_registry()*. You can take advantage of static initialization to populate the registry with your custom types. For example:

```
static int register_globals_ABC(json::Registry* registry){
  registry->register_object(
      parsefields_Foo, dumpfields_Foo);
  registry->register_object(
      parsefields_Bar, dumpfields_Bar);
  registry->register_object(
      parsefields_Baz, dumpfields_Baz);
  return 0;
}

static const int kDummy_ABC = register_globals_ABC(
  json::stream::global_registry());
```

In fact, this is exactly what is done by the macro invocation `JSON_REGISTER_GLOBALLY(ABC, Foo, Bar, Baz);`. If you want a macro to generate the function definition but not register it globally during static initialization, you can use the macro invocation `JSON_DEFN_REGISTRATION_FN(ABC, Foo, Bar, Baz;`. Both of these macros can be prefixed by *static* to make the corresponding function static.

2.4 Using the json_gen program

In addition to C-Preprocessor macros to generate parsers, dumpers and registration functions, there is a python program *json_gen.py* that can generate them for you. The usage is:

```
usage: json_gen.py [-h] [-o OUTDIR] [-b BASENAME] infiles [infiles ...]

Generate headers/sources for json-serializable streaming interface. This does
the same thing as the JSON_DECL and JSON_DEFN macros but allows for more
readable debugging as the source code isn't hidden beneath the preprocessor
macros.

positional arguments:
  infiles               specification files to process

optional arguments:
  -h, --help            show this help message and exit
  -o OUTDIR, --outdir OUTDIR
                        directory where to put output files. Default is cwd
  -b BASENAME, --basename BASENAME
                        basename of output files. Default is basename of input
                        file.
```

The input files are written in python and are declarative in nature. There are only a couple of functions that are needed:

```
def set_options(
   include_global_registration=True,
   emit_static_functions=False,
   namespaces=None,
   registration_suffix=None):
"""
```

(continues on next page)

(continued from previous page)

```
Set additional options:
 * include_global_registration: if true, the custom type will be
   registered with the global registry during static initialization
 * emit_static_functions: if true, a header file is not generated and
   the parse/dump and registration functions will be declared static
 * namespaces: a list of namespaces under which the generated functions
   should be named.
"""

def add_header_includes(list_of_headers):
"""
Add to the list of headers to include in the generated header file.
"""

def add_source_includes(list_of_includes):
"""
Add to the list of headers to include in the generated source file.
"""

def decl_json(spec):
"""
<spec> is a dictionary where keys are type declarations
(e.g. "Foo", "foo::Bar::Baz", "decltype(FooBar::z)") which map to
a list of strings that denote the fields which should be serialized for
that type.

It will generate a pair of functions parsefields_<suffix> and
dumpfields_<suffix> implementing the parse/dump API for a custom type
specified by <decl>. <suffix> is generated from the string <decl>.
"""
```

For example, consider the following:

```
// file: foo.h
#pragma once
struct Bar {
   int field_c;
   std::string field_d;
};

struct Foo {
   int field_a;
   Bar field_b;
};
```

Then we might create the following specification file:

```
# file: json_gen_foo.py
set_options(registration_suffix="Foo")
add_header_includes(["foo.h"])
decl_json({
 "Foo": ["field_a", "field_b"],
 "Bar": ["field_c", "field_d"]})
```

And we would execute:

```
$ python /path/to/json/json_gen.py json_gen_foo.py
```

Which would generate two files:

```
// json_gen_foo.h
# pragma once
#include "json/json.h"
#include "json/type_registry.h"
#include "foo.h"

int parsefields_Foo(
    const json::stream::Registry& registry, const re2::StringPiece& key,
    json::LexerParser* stream, Foo* out);
int dumpfields_Foo(
    const Foo& value, json::Dumper* dumper);

int parsefields_Bar(
    const json::stream::Registry& registry, const re2::StringPiece& key,
    json::LexerParser* stream, Bar* out);
int dumpfields_Bar(
    const Bar& value, json::Dumper* dumper);
```

```
// json_gen_foo.cc
#include "json_gen_foo.h"

int parsefields_Foo(
    const json::stream::Registry& registry, const re2::StringPiece& key,
    json::LexerParser* stream, Foo* out){
  uint64_t keyid = json::RuntimeHash(key);
  switch(keyid){
    case json::Hash("field_a"):
      return registry.parse_value(stream, &out->field_a);
    case json::Hash("field_b"):
      return registry.parse_value(stream, &out->field_b);
    default:
      json::sink_value(stream);
      return 1;
  }
  return 0;
}

int dumpfields_Foo(
    const Foo& value, json::Dumper* dumper){
  int result = 0;
  result |= dumper->dump_field("field_a", value.field_a);
  result |= dumper->dump_field("field_b", value.field_b);
  return result;
}

int parsefields_Bar(
    const json::stream::Registry& registry, const re2::StringPiece& key,
    json::LexerParser* stream, Bar* out){
  uint64_t keyid = json::RuntimeHash(key);
  switch(keyid){
    case json::Hash("field_c"):
      return registry.parse_value(stream, &out->field_c);
    case json::Hash("field_d"):
```

(continues on next page)

(continued from previous page)

```
      return registry.parse_value(stream, &out->field_d);
    default:
      json::sink_value(stream);
      return 1;
  }
  return 0;
}

int dumpfields_Bar(
    const Bar& value, json::Dumper* dumper){
  int result = 0;
  result |= dumper->dump_field("field_c", value.field_c);
  result |= dumper->dump_field("field_d", value.field_d);
  return result;
}

int register_types_Foo(json::Registry* registry){
  registry->register_object(
      parsefields_Foo, dumpfields_Foo);
  registry->register_object(
      parsefields_Bar, dumpfields_Bar);
  return 0;
}

static const int kDummy_Foo = register_types_Foo(json::global_registry());
```

2.5 Implementation Details

The type registry allows us to store function pointers to each of the conversion functions for a specific type. The parse and dump implementations are based on function templates to do most of the work, but the field enumerators and tokenization functions are looked up at runtime.

This is done by associating a unique numeric key with each type that is registered in the type registry. The key is actually the address of a template function instanciation (`json::stream::sentinel_function`). The function has an empty body and exists just to help the program generate unique keys for each type.

The type keys are determined when the program is linked (when the function address is determined). The parse/dump implementation functions are added to the registry during static initialization (the function pointers are stored in a map using the type key). Then during the parse/dump dispatch they are retrieved from the registry using the same key.

The implementation is in *type_registry.h* and *type_registry.cc*.

CHAPTER 3

Stream API (deprecated)

Warning: This document describes the old experimental streaming API included in version 0.1.0. That API has been replaced in version 0.2.0 with a new registry based implementation. This document is retained for historical purposes. A section at the end describes some of the drawbacks of this approach and why it was replaced.

The "stream" API is meant to provide a mechanism for creating high level bindings for parsing a JSON character buffer directly into static objects. You can use it to make your C/C++ structures JSON-serializable.

The JSON event stream is provided by the `LexerParser` which maintains a state machine for the tokenization and semantic parsing of a JSON text document. It provides the `GetNextEvent()` which sequentially returns each semantic event in the stream.

3.1 Parsing

Parsing JSON entities into C++ entities is generally done through overloads of the `ParseValue()` function, which has the following signature:

```
void ParseValue(const Event& event, LexerParser* stream, ValueType* value);
```

for each parseable `ValueType`. `ParseValue()` is essentially a dispatcher which dispatches either a token parser (for scalar types), an object parser (for object types) or an array parser (for array types).

For scalar types, the implementation generally looks something like this, where it dispatches a token parser:

```
void ParseValue(const Event& event, LexerParser* stream, Foo* value) {
  if (event.typeno != json::Event::VALUE_LITERAL) {
    LOG(WARNING) << fmt::format("Cannot parse {} as value at {}:{}",
                                json::Event::ToString(event.typeno),
                                event.token.location.lineno,
                                event.token.location.colno);
  }
```

(continues on next page)

(continued from previous page)

```
  ParseToken(event.token, value);
}
```

For C/C++ structures and classes, the implementation generally looks like this:

```
void ParseValue(const Event& event, LexerParser* stream, Foo* out) {
  ParseObject(event, stream, out);
}
```

`ParseObject()` is a function template which simply does the following:

1. Ensures that the parse stream matches the expected sequence:
    a. Starting with `OBJECT_BEGIN`
    b. Followed by a sequence of *(OBJECT_KEY, value)* pairs
    c. Ending with `OBJECT_END`
2. Iterates through each of the *(OBJECT_KEY, value)* pairs and calls `ParseField()` on each pair.

`ParseField()` is, in turn, overloaded for every serializable type. The purpose of `ParseField()` is just to select which member to call `ParseValue()` on, given the current JSON object key. The implementation generally looks like this:

```
int ParseField(const re2::StringPiece& key, const Event& event,
               LexerParser* stream, Foo* out) {
  uint64_t keyid = RuntimeHash(key);
  switch (keyid) {
    case Hash("field_a"):
      ParseValue(event, stream, &out->field_a);
      break;
    case Hash("field_b"):
      ParseValue(event, stream, &out->field_b);
      break;
    case Hash("field_c"):
      ParseValue(event, stream, &out->field_c);
      break;
    default:
      SinkValue(event, stream);
      return 1;
  }
  return 0;
}
```

In summary, the logic follows this call-map:

```
ParseValue
  ├─ ParseToken
  └─ ParseObject (template)
     └─ ParseField (overload)
        └─ ParseValue (overload)
```

All JSON-parsable types must implement `ParseValue()`. JSON-parsable scalars my utilize `ParseToken()` if an overload exists, or they may implement the token parser directly in `ParseValue()`. JSON-parsable objects must implement `ParseValue()` as a single-line function call to `ParseObject()` and must also implement `ParseField()`.

Note that `ParseValue()` overloads are necessary mostly due to the static nature of C++. You could imagine an implementation that looks like the following:

```
void ParseValue(const Event& event, LexerParser* stream, Foo* value) {
  if (event.typeno == json::Event::BEGIN_OBJECT) {
    ParseObject(event, stream, value);
  } elif (event.typeno == json::Event::BEGIN_ARRAY) {
    ParseArray(event, stream, value);
  } else {
    ParseScalar(event.token, value);
  }
}
```

But for a given type `Foo` only one of these functions will have applicable overloads. The other two wont exist and we'll get compiler errors. This is why `ParseValue()` needs to be overloaded for every type.

3.2 Problems with two-stage lookup

The above strategy is problematic due to the idosyncracies of two-stage lookup in C++. In order to compile successfully, the template definition for `ParseObject()` must come after the declaration for all `ParseValue()` overloads in the translation unit where it is instanciated for a particular type. This leads to the ugly pattern of:

1. Include the basic stream headers
2. Declare all the overloads (or include the headers that declare them)
3. Include the "stream_tpl.h" header, which has the implementation of the helper templates.
4. Define the overloads.

This means that correct compilation of any translation unit depends on a strict and esoteric order of the includes. It also requires including a header *after* some C++. This could be mitigated by hiding the template definition within a macro, but then it would require a magic macro to drop a template definition somewhere in the source file. And, again, the placement of that macro call will be specific and esoteric.

All of this leads to a poor library user experience and does not provide a good mechanism for extensibility. Starting with version 0.2.0 a new pluggable scheme is implemented which avoids these pitfalls.

CHAPTER 4

The json program

Included in the package is a simple json utility application intended to demonstrate usage of the library. The command *json* can dump the lex'ed token stream or the parsed event stream. It can also validate a json file or markup its contents with html that can be used to publish semantic-highlighted json documents.

```
====
json
====
version: 0.2.0
author : Josh Bialkowski <josh.bialkowski@gmail.com>
copyright: (C) 2018

json [-h/--help] [-v/--version] <command>

Demonstrates the usage of the json library to lex and parse JSON data

Flags:
------
-h  --help          print this help message
-v  --version       print version information and exit

Positionals:
------------
command             Each subcommand has it's own options and arguments, see
                    individual subcommand help.

Subcommands:
------------
lex                 Lex the file and dump token information
markup              Parse and dump the contents with HTML markup
parse               Parse the file and dump actionable parse events
verify              Parse the file and exit with 0 if it's valid json
```

CHAPTER 5

Changelog

5.1 v0.2.6 – in-progress

- Implement travis build
- Implement pseudo-release-tag -> readthedocs build pipeline
- Remove monorepo dependencies from the sparse export

Closes: adfd69d, c3f0854, 296d380

5.2 v0.2.5

- Tokenizer and Parser now have peek-ahead capability
- Don't choke on empty list ("[]") or object ("{}")
- Fix sink_object and sink_list were inconsistent in whether or not they expected the opening event to have been consumed or not
- Add parse utilities for parsing lists into standard containers with push_back members
- Add parse API for lists mirroring the parse API for objects
- Fix *sink_object* sign error in loop condition
- Add string type name to the registry entry
- Add stream specialization for *shared_ptr*

5.3 v0.2.4

- change all functions to GNU style snake_case

5.4 v0.2.3

- get rid of WalkOpts
- remove registry_poc.cc
- get rid of BufPrinter and FmtError, replace with static stringstream
- make Registry* const within the dumper
- parse_array -> parse_list
- make order of parameters consistent in type_registry.h API
- parse functions all return integers
- some cleanup/reorg of type_registry.h/cc

5.5 v0.2.2

- Use SerializeOpts inside StreamDumper to pretty-print (ish)
- convert json_gen to generate code for the new registry-based streaming API and update stream_gen_test to use it
- change stream_macros.h to generate code for the new registry-based streaming API and update stream_test to use it.
- add kCompactOpts to json.h/cc (for unpretty-printing)
- add documentation for the new streaming API and some general docs
- delete old streaming API

5.6 v0.2.1

- Added generic tree walk to the stream API, allows arbitrary navigation of json-serializable structures
- Add python script to code-generate the stream API rather than using C macros.
- Add utilities to escape/unescape strings for JSON serialization.
- Fix missing backslash in regex for STRING_LITERAL token
- Cleanup some compiler warnings
- Add a frontend test to execute the demo program on some canonical input and ensure that the lex/parse/markup output matches expected outputs
- ParseString will now unescape the contents
- Moved parse/emit functions to their own files
- Moved parse/emit functions out of the stream namespace
- Merge _tpl.h files into -> .h
- Make json_gen use a jinja template
- Replace remaining printf() with LOG() for parse errors

5.7 v0.2.0

Overhaul the stream API.

- Stream API no longer uses runtime pointer maps
- Implement compile time string hashing for key switch/case
- Implement new macro technique for variable number of case statements
- Emit/Parse are now implemented as overloads in json::stream:: namespace rather than member functions of the struct. This may change again in the future.

5.8 v0.1.0

Initial commit.

- Functional low-level API for lexing/parsing JSON
- A demonstrator "stream" API for creating JSON-serializable structures in C++.

CHAPTER 6

Release Notes

6.1 v0.2 series

6.1.1 v0.2.6

This release adds a bunch of build tooling, but nothing in the library has changed.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search