mapry Documentation

Release 1.0.0.4

Marko Ristin

Jun 02, 2021

CONTENTS:

1 Introduction 1

2 Design Decisions 5

3 Schema 7

4 C++ Specifics 17

5 Go Specifics 23

6 Python Specifics 27

7 Installation 33

8 Command-Line Usage 35

9 mapry Module 37

10 Related Projects 45

11 Future Work 47

12 Contributing 49

13 Versioning 51

14 Indices and tables 53

Python Module Index 55

Index 57

CHAPTER

ONE

INTRODUCTION

Mapry generates polyglot code for de/serializing object graphs from JSONable structures.

Story. We needed a yet another domain-specific language for internal data exchange and configuration of the system. The existing solutions mostly focused on modeling the configuration as *object trees* in which the data is nested in hierarchies with no **cross-references** between the objects.

For example, think of object trees as JSON objects or arrays. We found this structure to be highly limiting for most of the complex messages and system configurations. Our use cases required objects in the data to be referenced among each other – instead of object trees we needed **object graphs**.

Moreover, we wanted the serialization itself to be **readable** so that an operator can edit it using a simple text editor. JSONable structure offered itself as a good fit there with a lot of existing assistance tools (JSON and YAML modules *etc*.).

However, JSON allows only a limited set of data types (numbers, strings, arrays and objects/maps). We found that most of our data relied on **a richer set of primitives** than was provided by a standard JSON. This extended set includes:

- date,
- datetime,
- time of day,
- time zone,
- duration and
- path.

While there exist polyglot serializers of object trees (*e.g.*, Protocol Buffers), language-specific serializers of object graphs (*e.g.,* Gob in Go or Pickle in Python) or polyglot ones with a limited set of primitives (*e.g.,* Flatbuffers), to the best of our knowledge there is currently no serializer of **object graphs** that operates with **readable representations** and provides a **rich set of primitive data types** consistently **across multiple languages**.

Hence we developed Mapry, a generator of polyglot de/serialization code for object graphs from JSONable structures.

The **schema** of the object graph is stored in a separate JSON file and defines all the data types used in the object graph including the object graph itself. The code is generated based on the schema. You define schema once and generate code in all the supported languages automatically. Schemas can be evolved and backward compatibility is supported through optional properties.

1.1 Supported languages

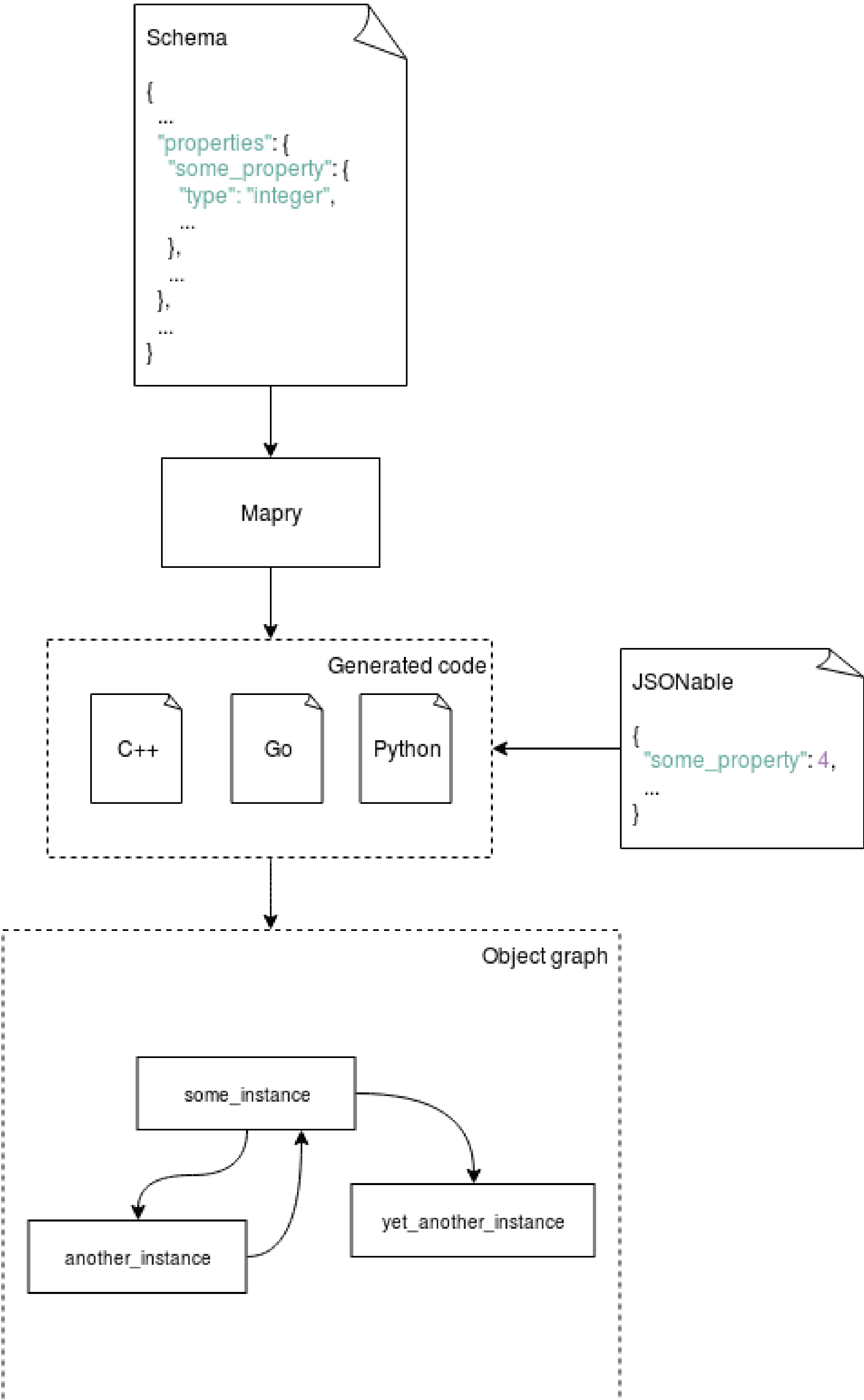
Currently, Mapry speaks:

- C++11,
- Go 1 and
- Python 3.

Since the serialization needs to operate in different languages, only the intersection of language features is supported. For example, since Go does not support inheritance or union types, they are not supported in Mapry either.

1.2 Workflow

The following diagram illustrates the workflow.


Schema
{
...
"properties": {
"some_property": {
"type": "integer",
...
},
...
},
...
}
Mapry
Generated code
C++
Go
Python
JSONable
{
"some_property": 4,
...
}
Object graph
some_instance
another_instance
yet_another_instance

CHAPTER TWO

DESIGN DECISIONS

Maintainability. We wanted to facilitate maintainability of the system through as many static and run time checks as possible so that most errors in the object graphs are registered prior to the deployment in the production. These checks include strong typing annotations at generation time and various runtime checks at deserialization (dangling references, range checks, minimum number of elements in arrays, pattern matching *etc.*).

Versatility. Since we need humans to operate on object graphs, we needed the data representation of the object graph to be readable and editable. Hence we strived to make the resulting JSONable structures succinct yet comprehensible.

We intentionally did not fixate Mapry to directly handle files to allow for a larger variety of supported formats and sources (JSON, YAML, BSON, MongoDB *etc.*). Mapry operates on an in-memory representation of the JSONable data (such as Python dictionaries or Go `map[string]interface{}`) which makes it much more versatile than if it handled data sources directly.

Code readability over speed. We wanted the generated code to be rather readable than fast. Though the seasoned developers might not care about the implementation details, we found that newcomers really like to peek under the hub. They get up to speed much faster when the generated code is readable.

In particular, when the generated code permeates most of your system components, the readability becomes a paramount when you fix bottlenecks or debug.

Avoid dependency hell. We explicitly decided to make the generated code as stand-alone as possible. This includes generating redundant data structures such as parsing errors which could be theoretically used across different generated modules.

While this redundancy seems wasteful (duplication) or impractical (specific errors need to be checked instead of general ones), stand-alone code allows us to dispense of a common Mapry library which greatly alleviates the dependency hell in larger systems.

Take Protocol buffers as a contrasting example. The code generated by protocol buffers depends on a common `protobuf` library. Imagine you depend on two different libraries, each using a different version of protocol buffers. Since your system now has conflicting dependencies, there is usually no easy way to use both libraries in the system. If you are not the owner, you need to contact the maintainers of one of the libraries and ask them for an upgrade.

CHAPTER

THREE

SCHEMA

The Mapry schema defines the properties and structures of the object graph in a single JSON file. This file is parsed by Mapry to generate the dedicated de/serialization code in the respective languages.

The schema is split in following sections:

- graph name and description,
- language-specific settings,
- definition of composite structures (classes and embeddable structures, see below) and
- definition of graph properties.

3.1 Introductory Example

Before we dwell into the separate sections, let us present a brief example of a schema to give you an overview:

```
{
  "name": "Pipeline",
  "description": "defines an address book.",
  "cpp": {
    "namespace": "book::address",
    "path_as": "boost::filesystem::path",
    "optional_as": "std::experimental::optional",
    "datetime_library": "ctime"
  },
  "go": {
    "package": "address"
  },
  "py": {
    "module_name": "book.address",
    "path_as": "pathlib.Path",
    "timezone_as": "pytz.timezone"
  },
  "classes": [
    {
      "name": "Person",
      "description": "defines a contactable person.",
      "properties": {
        "full_name": {
          "type": "string",
          "description": "gives the full name (including middle names)."
        },
```

(continues on next page)

(continued from previous page)

```
        "address": {
          "type": "Address",
          "description": "notes where the person lives.",
        }
      }
    }
  ],
  "embeds": [
    {
      "name": "Address",
      "description": "defines an address.",
      "properties": {
        "text": {
          "type": "string",
          "description": "gives the full address."
        }
      }
    }
  ],
  "properties": {
    "maintainer": {
      "type": "Person",
      "description": "indicates the maintainer of the address book."
    }
  }
}
```

3.2 Language-specific Settings

Language-specific settings instruct Mapry how to deal with non-standard structures during code generation. For example, you need to instruct which path library to use in Python to represent file system paths (`str` or `pathlib.Path`). Note that settings can be specified only for a subset of languages. For example, you can omit C++ settings if you are going to generate the code only in Go and Python.

The available settings are explained for each language in the *C++ Specifics*, *Go Specifics* and *Python Specifics*, respectively.

3.3 Data Types

Mapry defines three families of data types: primitive data types (*primitives*), aggregated data types (*aggregated*) and composite data types (*composites*).

Primitives are the basic data types such as booleans and integers.

Aggregated represent data structures which contain other data structures as values. Mapry provides two aggregated data types: arrays and maps.

Composites represent data structures which contain **properties**. Each property of a composite has a name and a corresponding data type. Three types of composites are available in Mapry: classes, embeddable structures and an object graph.

The following subsections describe the data types, instruct you how to define them in the schema and how to impose constraints to further specify them. For implementation details in different languages, please consult: *C++ Specifics*, *Go Specifics* and *Python Specifics*.

For a summary of how Mapry represents the data types described below, see *JSON Representation*.

3.3.1 Primitive Types

boolean designates a value can be either true or false.

Booleans are represented in Mapry as JSON booleans.

integer defines an integer number.

You can constrain integers by a minimum and maximum properties (`minimum` and `maximum`, respectively). Following JSON schema, Mapry assumes inclusive limits, unless you specify them otherwise with boolean `exclusive_minimum` and `exclusive_maximum` properties.

Integers are represented as JSON numbers.

Note that different languages can represent different (and mutually possibly incompatible!) ranges of integers. See *Numbers in C++*, *Numbers in Go* and *Numbers in Python* for more details.

float specifies a floating-point number.

Analogous to integers, floating-point numbers can be further constrained by minimum and maximum properties (`minimum` and `maximum`, respectively). These limits are tacitly assumed inclusive. You can specify exclusive limits by setting `exclusive_minimum` and/or `exclusive_maximum` properties to true, respectively.

Floating-point numbers are represented as JSON numbers.

Note that different languages can represent different (and mutually possibly incompatible!) ranges of floating-point numbers. See *Numbers in C++*, *Numbers in Go* and *Numbers in Python* for more details.

string denotes a string of characters.

You can enforce a string to follow a regular expression by defining the `pattern` property.

Mapry represents strings as JSON strings.

path represents a path in a file system.

Similar to strings, paths can also be restricted to comply to a regular expression by specifying the `pattern` property.

Paths are represented as JSON strings in Mapry.

date designates a day in time.

The time zone is not explicitly given, and needs to be assumed implicitly by the user or specified separately as a related time zone value (see below).

The dates are represented as JSON strings and expected in ISO 8601 format (*e.g.*, `"2016-07-03"`). If your dates need to follow a different format, you need to specify the `format` property. Supported format directives are listed in *Date/time Format*.

time ticks a time of day.

Mapry represents time of day as JSON strings and assumes them by default in ISO 8601 format (*e.g.*, `"21:07:34"`). However, you can change the format by setting the `format` property. For a list of available format directives, see *Date/time Format*.

datetime fixes an instant (time of day + day in time).

Parallel to `date`, the datetime does not explicitly assume a time zone. The user either presumes the zone by a convention or specifies it as a separate time zone value (see below).

Just as `date` and `time` so is `datetime` represented as JSON string in ISO 8601 format (*e.g.*, `"2016-07-03T21:07:34Z"`) implicitly assuming UTC time zone. If you want to have a datetime value in a different format, you have to set the `format` property. See *Date/time Format* for a list of format directives.

`time_zone` pins a time zone.

Time zone values are useful as companion values to date and datetimes.

Mapry represents time zones as JSON strings, identified by entries in IANA time zone database.

For example, `"Europe/Zurich"`

`duration` measures a duration between two instants.

Durations can be both positives and negatives. Following C++ std::chrono library, Mapry assumes a year as average year (365.2425 days) and a month as average month (30.436875 days). If a duration should denote actual months from a given starting date, you have to represent the duration as strings and manually parse them by a third-party library (*e.g.*, isodate in Python).

For example, `"P6M2.1DT3H54M12.54S"` (6 months, 2.1 days, 3 hours, 54 minutes and 12.54 seconds).

Note that different languages can represent different (and mutually possibly incompatible!) granularities and ranges of durations. See *Durations in C++*, *Durations in Go* and *Durations in Python* for more details.

3.3.2 Aggregated Types

`array` lists an ordered sequence of values.

Mapry arrays are strongly typed and you need to specify the type of the values as `values` property.

The minimum and maximum size of the array (inclusive) can be further specified with the properties `minimum_size` and `maximum_size`, respectively.

If you need to capture tuples, you can define an array of both minimum and maximum size set to the same number.

Arrays are represented as JSON arrays.

`map` projects strings to values (in other words, indexes values by strings).

Map values are strongly typed in Mapry and need to be defined as `values` property.

Mapry represents maps as JSON objects.

3.3.3 Composite Types

Primitive and aggregated data types are the building blocks of a Mapry schema. They are further structured into **classes** and **embeddable structures**. Think of these structures as floors or building units. The whole building is further represented as an **object graph**, the encompassing data type. Composite data types are defined by their **properties**. All composite data types must be given names.

Mapry represents instances of composite data types as JSON objects where properties of the JSON object correspond to the properties defined for the composite data type.

Classes are referencable composite data types. Each instance of a class has a unique identity which serves as a reference.

Classes are defined as a list of objects as `classes` property of the schema. The order of the definitions is mimicked in the generated code as much as possible.

Each class needs to define the `name` and `description`. The plural form of the class instances can be specified as `plural`. If no plural is specified, Mapry infers the plural form using a heuristic. A class can define an `id_pattern`, a regular expression, which mandates the pattern of the instance identifiers of the class.

The properties of the class are specified as `properties` property of the class definition in the schema. See below how `properties` are defined. If a class defines no properties then `properties` can be omitted.

Embeddable structures are nested within other composite data types.

Embeddable structures are given as a list of objects as `embeds` property of the schema. The order of the definitions matters, and Mapry tries to follow it when generating the code.

Each embeddable structure needs to specify its `name` and `description`. Properties, if any, are given as `properties` property of the definition. See below how `properties` are specified.

Graph object is the encompassing data type corresponding to the schema.

The graph object needs to have a `name` and a `description`.

The properties of graph object itself, if available, are defined as `properties` property of the schema.

The classes and embeddable structures are defined as `classes` and `embeds` properties of the schema, respectively.

Properties are the essence of composite data types. The `properties` of a composite type (be it class, embeddable structure or graph object) map property names to property definitions given as JSON objects in the schema.

A property type can be either a primitive data type, an aggregated data type, a reference to a class or a nested embeddable structure. The type of the property is given as `type` property of the property definition. The `type` corresponds either to the name of the primitive, aggregated or composite type.

Each property must have a `description` written as a JSON string in the property definition.

Properties are assumed mandatory by default. You can specify that a property is optional by setting the `optional` to true in the property definition. Mapry will raise an error when parsing a JSONable object representing the composite which lacks a mandatory property. On the other side, optional properties can be simply omitted in the JSONable. If you need to evolve a schema over time, optional properties provide you a practical approach to handle different versions of a composite.

Mapry uses a heuristic to determine the property name in the JSONable object representing the composite (see *JSON Representation*). In most cases, you can leave Mapry decide the property names for you. However, you can specify a different property name of the respective JSONable by setting `json` property in the property definition if for some reason you need to evolve the name or need to follow an external convention incompatible with the heuristic.

The additional constraints of primitive and aggregated types (such as minimum value of an integer or minimum size of an array) are given as additional properties in the property definition.

3.3.4 JSON Representation

Mapry represents an instance of an object graph as a JSON object. Properties of the object graph are directly represented as properties of that JSON object. While this works for unreferencable data types (primitive and aggregated data types and embeddable structures), instances of the classes need a special treatment.

Namely, instances of the classes are represented in an *instance registry* as JSON objects. Each property of the instance registry corresponds to an instance of the class: the identifier is the property name (*i.e.* a key), while the instance is the property value given as a nested JSON object (*i.e.* a value, with properties of that nested JSON object corresponding to the properties of the class).

Each instance registry is given as an additional property of the object graph. The name of the instance registries corresponds to the lowercase `plural` property of the class (if no `plural` is given, then the name of the instance registry is inferred by a heuristic).

References to an instance of a class in an object graph are given as JSON strings.

The following table summarizes how individual types are represented in JSONables.

Mapry Type	JSON Type	JSON Example
boolean	boolean	`true`
integer	number	`2016`
float	number	`198.4`
string	string	`"some text"`
path	string	`"/a/path/to/somewhere"`
date	string	`"2016-07-03"`
time	string	`"21:07:34"`
datetime	string	`"2016-07-03T21:07:34Z"`
time_zone	string	`"Europe/Zurich"`
duration	string	`"P2DT3H54M12.54S"`
array	array	`[1, 2, 3]`
map	object	`{"someKey": 1, "anotherKey": 3}`

embeddable structure	object	`{`
		`    "someProperty": 23,`
		`    "anotherProperty": "some text"`
		`}`

instance of a class	object	`{`
		`    "someProperty": 23,`
		`    "anotherProperty": "some text"`
		`}`

reference to an instance of a class	string	`"some-id"`

object graph	object	`{`
		`    "persons": {`
		`        "Alice": {`
		`            "birthday": "2016-07-03",`
		`            "bff": "Bob"`
		`        },`
		`        "Bob": {`
		`            "birthday": "2015-03-21"`
		`        },`
		`        "Chris": {`
		`            "birthday": "2016-11-15",`
		`            "bff": "Bob"`
		`        }`
		`    },`
		`    "maintainer": "Bob"`
		`}`

3.4 Date/time Format

Representation of date/times in Mapry matches ISO 8601 by default (`"2016-07-03"`, `"21:07:34"` and `"2016-07-03T21:07:34Z"`). This works perfectly fine for cases where you control the data and can assume that the reference time zone is UTC. Yet when you do not control the data, *e.g*, when it comes from external sources, you need to adapt the expected date/time format.

Mapry allows you to specify the format of a date/time through `format` constraint consisting of widely-used strptime/strftime directives (*e.g.*, see strftime and strptime behavior in Python and ctime strftime in C++). Since code needs to be generated in multiple languages and not all languages support all the directives, only a subset of the directives are available:

Diractive	Description
`%a`	The abbreviated weekday name ("Sun")
`%A`	The full weekday name ("Sunday")
`%b`	The abbreviated month name ("Jan")
`%B`	The full month name ("January")
`%d`	Day of the month (01..31)
`%e`	Day of the month with a leading blank instead of zero (1..31)
`%m`	Month of the year (01..12)
`%y`	Year without a century (00..99)
`%Y`	Year with century
`%H`	Hour of the day, 24-hour clock (00..23)
`%I`	Hour of the day, 12-hour clock (01..12)
`%l`	Hour of the day, 12-hour clock without a leading zero (1..12)
`%M`	Minute of the hour (00..59)
`%P`	Meridian indicator ("am" or "pm")
`%p`	Meridian indicator ("AM" or "PM")
`%S`	Second of the minute (00..60)
`%z`	Time zone hour and minute offset from UTC
`%Z`	Time zone name
`%%`	Literal `%` character

For example, you can match month/day/year format with `%m/%d/%Y` or day.month.year. hours:minutes:seconds with `%d. %m. %Y %H:%M:%S`.

Mapry-generated code will use the standard date/time libraries (unless otherwise specified in the language-specific settings). This means that the implementation of the library determines how the directives are interpreted, which could be sometimes ambiguous or not straight-forward to understand. For example, time zone information (`%z` and `%Z` directives) might be handled differently by different implementations.

Additionally, due to lack of escaping in Go standard package `time`, certain formats can not be handled. See *Go Date/time Format Directives* for details.

3.5 Conventions

Since Mapry needs to generate code in different languages, parts of the schema such as property names and class descriptions need to follow certain conventions to comply with the readability and style rules of the corresponding languages.

Names of the object graph, classes and embeddable structures are expected as `Sanke_case`. Abbreviations are expected as upper-case, *e.g.*, `Some_IDs` or `Some_URLs`.

Property names are generally expected in `snake_case` with first word lower-cased. Abbreviations are expected in upper-case even at the beginning of a property name: `some_IDs`, `some_URLs`, `IDs_to_store`, `URLs_to_fetch`.

Descriptions should all end with a dot and start with a lower-case. The descriptions should start with a lower-case verb in present tense, *e.g.*, `indicates the maintainer of the address book.`

3.6 Further Examples

The brief example presented in *Introductory Example* gives you only an overview and lacks a comprehensive collection of use cases. To further demonstrate how to define the object graphs and how they are represented as JSONables in many different scenarios, we provide the following table with links to examples.

Description	Schema	Example representation
boolean	schema	JSON file
integer	schema	JSON file
float	schema	JSON file
string	schema	JSON file
path	schema	JSON file
date	schema	JSON file
time	schema	JSON file
datetime	schema	JSON file
time_zone	schema	JSON file
duration	schema	JSON file
array	schema	JSON file
map	schema	JSON file
embeddable structure	schema	JSON file
class instances	schema	JSON file
optional property	schema	JSON file
differing JSON property	schema	JSON file

For yet more examples, please see the remainder of the test cases. Each test case consists of a schema (`schema.json`), generated code (`{language}/test_generate` subdirectory) and example JSON representations (`example_ok.json`, `example_ok_*.json` and `example_fail_*.json`).

CHAPTER FOUR

C++ SPECIFICS

Mapry produces a C++ runtime implementation with a simple interface for de/serializing object graphs from Jsoncpp values.

4.1 Settings

You need to specify the C++ specific settings in a schema to instruct Mapry how to generate the code. The following points needs to be defined:

namespace indicates the namespace of the generated code.

For example, `book::address`.

path_as defines the type of the paths in the generated code.

Mapry supports: `std::filesystem::path` and `boost::filesystem::path`.

optional_as defines the type of the optional properties in the generated code.

Mapry supports: `boost::optional`, `std::optional` and `std::experimental::optional`.

datetime_library defines the library to use for date, datetime, time and time zone manipulation.

Mapry supports: `ctime` and `date.h` (*i.e.* Howard Hinnant's date library)

idention defines the indention of the generated code. Defaults to two spaces and can be omitted.

For example, `"    "` (four spaces)

4.2 Generated Code

Mapry produces all the files in a single directory. The generated code lives in the namespace indicated by C++ setting `namespace` in the schema.

Mapry generates the following files (in order of abstraction):

- `types.h` defines all the graph structures (embeddable structures, classes, object graph itself *etc.*).
- `parse.h` and `parse.cpp` define the structures used for parsing and implement their handling (such as parsing errors).
- `jsoncpp.h` and `jsoncpp.cpp` define and implement the de/serialization of the object graph from/to a Jsoncpp value.

The example of the generated code for the schema given in *Introductory Example* is available in the repository.

4.3 Deserialization

The following snippet shows you how to deserialize the object graph from a Jsoncpp value. We assume the schema as provided in *Introductory Example*.

```
Json::Value value;
// ... parse the value from a source, e.g., a file

book::address::parse::Errors errors(1024);
book::address::Pipeline pipeline;

const std::string reference_path(
    "/path/to/the/file.json#");

book::address::jsoncpp::pipeline_from(
    value,
    reference_path,
    &pipeline,
    &errors);

if (not errors.empty()) {
    for (const auto& err : errors.get()) {
        std::cerr << err.ref << ": " << err.message << std::endl;
    }
    return 1;
}
```

You can seamlessly access the properties and iterate over aggregated types:

```
std::cout << "Maintainers are:" << std::endl;
for (const book::address::Person& maintainer : pipeline.maintainers) {
    std::cout
        << maintainer.full_name
        << " (address: " << maintainer.address.text << ")"
        << std::endl;
}
```

4.4 Serialization

You serialize the graph to a Jsoncpp value (assuming you predefined the variable `pipeline`) simply with:

```
const Json::Value value(
        book::address::jsoncpp::serialize_pipeline(
            pipeline));
```

4.5 Compilation

The generated code is *not* header-only. Since there is no standard C++ build system and supporting the whole variety of build systems would have been overly complex, we decided to simply let the user integrate the generated files into their build system manually. For example, Mapry will *not* generate any CMake files.

Here is an exerpt from a `CMakeLists.txt` (corresponding to the schema given in *Introductory Example*) that uses conan for managing dependencies:

```
add_executable(some_executable
    some_executable.cpp
    book/address/types.h
    book/address/parse.h
    book/address/parse.cpp
    book/address/jsoncpp.h
    book/address/jsoncpp.cpp)

target_link_libraries(some_executable
    CONAN_PKG::jsoncpp
    CONAN_PKG::boost)
```

4.6 Implementation Details

4.6.1 Representation

Mapry represents the types defined in the schema as closely as possible in C++. The following tables list how different types are represented in generated C++ code.

Table 1: Primitive types

Mapry type	C++ type
Boolean	`bool`
Integer	`int64_t`
Float	`double`
String	`std::string`
Path	`std::filesystem::path` or `boost::filesystem::path` (depending on `path_as` setting)
Date	`struct tm` or `date::local_days` (depending on `datetime_library` setting)
Time	`struct tm` or `date::time_of_day<std::chrono::seconds>` (depending on `datetime_library` setting)
Datetime	`struct tm` or `date::local_seconds` (depending on `datetime_library` setting)
Time zone	`std::string` or `const date::time_zone*` (depending on `datetime_library` setting)
Duration	`std::chrono::nanoseconds`

Table 2: Aggregated types (of a generic type T)

Mapry type	C++ type
Array	`std::vector<T>`
Map	`std::map<std::string, T>`

Table 3: Composite Types

Mapry type	C++ type
Reference to an instance of class T	`T*`
Embeddable structure T	`struct T`
Optional property of type T	`boost::optional<T>`, `std::optional<T>` or `std::experimental::optional<T>` (depending on `optional_as` setting)

Table 4: Graph-specific structures

Mapry type	C++ type
Registry of instances of class T	`std::map<std::string, T>`

4.6.2 Numbers

Mapry depends on the underlying JSON library for the representation of numbers. How the library deals with numbers has implications on the ranges and precision of the numbers that you can represent and can lead to unexpected overflows.

While JSON standard does not distinguishes between integers and floats and treat all numbers equally, Jsoncpp indeed distinguishes between the integers (represented internally as 64-bit integers) and floats (represented internally as double-precision floats).

Based on the internal representation, C++ deserialization can represent integers in the range of 64-bit integers (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) and floats in the rage of double-precision (-1.7976931348623157e+308 to 1.7976931348623157e+308).

However, note that deserialization in other languages might impose stricter constraints. For example, Go does not distinguish between integers and floats when parsing JSON (see *Numbers in Go*), so the overall range that you can represent is smaller if you need Go and C++ de/serialization to inter-operate.

4.6.3 Time Libraries

Mapry generates the code that uses either the standard ctime library or Howard Hinnant's date library (date.h) to manipulate the dates, datetimes, times of the day and time zones based on `datetime_library` in C++ settings section of the schema.

Since `ctime` does not support a time zone registry, the time zones are parsed as strings and are not further validated. For example, you can specify an incorrect time zone such as `Neverland/Magic` and the deserialization code will not complain.

On the other hand, since Howard Hinnant's date library (date.h) supports a registry of IANA time zones, the time zones are in fact checked at deserialization and an error will be raised if the time zone is invalid.

We would recommend you to use Howard Hinnant's date library (date.h) instead of the standard `ctime` though it comes with an extra effort of installing the dependenciy. In our opinion, the sophistication, the easy and the clarity Howard Hinnant's library enforces on date/time manipulations pay off in long term.

The following table gives you a comparison of the generated codes:

Date `ctime`: schema and code

`date.h`: schema and code

Datetime `ctime`: schema and code

`date.h`: schema and code

Time of day `ctime`: schema and code

`date.h`: schema and code

Time zone `ctime`: schema and code

`date.h`: schema and code

4.6.4 Durations

Mapry uses standard std::chrono::nanoseconds to represent durations. According to the standard, this implies that beneath the hub a signed integral type of at least 64 bits is used to represent the count.

Since integral numbers of finite size are used for representation, the generated code can only deal with a finite range of durations. In contrast, Mapry durations are given as strings and thus can represent a much larger range of durations (basically bounded only on available memory space).

In fact, the problem is very practical and you have to account for it when you deal with long or fine-grained durations. For example, a duration specified as `P300Y` already leads to an overflow since 300 years *can not* be represented as nanoseconds with finite integral numbers of 64 bits. Analogously, `PT0.0000000001` can not be represent either since the precision of the duration goes beyond nanoseconds.

Note also that other languages impose stricter constraints. For example, Python uses microseconds to represent durations (see *Durations in Python*) and hence you need to restrict your durations to microsecond granularity if both Python and C++ de/serializations are needed.

CHAPTER

FIVE

GO SPECIFICS

Mapry generates a Go package with structures and several public functions for de/serialization from/to JSONable objects given as *interface{}*.

5.1 Settings

In order to generate the Go code, you need to specify the Go specific setting in the schema:

package indicates the package name of the generated code.

5.2 Generated Code

All the files are generated in a single directory. The code lives in the package indicated by the Go setting `package` of the schema.

Mapry writes the following files (in order of abstraction):

- `types.go` defines all the structures of the object graph (embeddable structures, classes, object graph itself *etc.*)
- `parse.go` defines general parsing structures and their handling (such as parsing errors).
- `fromjsonable.go` provides functions for parsing the object graph from a JSONable `interface{}` value.
- `tojsonable.go` gives you functions for serializing the object graph to a JSONable `interface{}` value.

The example of the generated code for the schema given in *Introductory Example* is available in the repository.

5.3 Deserialization

Assuming the schema provided in *Introductory Example*, you deserialize the object graph from a JSONable `interface{}` as follows.

```
var value interface{}
// ... parse the value from a source, e.g., a file

pipeline := &address.Pipeline{}
errors := address.NewErrors(0)

const referencePath = "/path/to/the/file.json#"
```

(continues on next page)

(continued from previous page)

```
address.PipelineFromJSONable(
    value,
    referencePath,
    pipeline,
    errors)

if !errors.Empty() {
    ee := errors.Values()
    for i := 0; i < len(ee); i++ {
        fmt.Fprintf(
            os.Stderr, "%s: %s\n",
            ee[i].Ref, ee[i].Message)
    }
    return 1
}
```

The access to the deserialized object graph `pipeline` is straight-forward:

```
fmt.Println("Maintainers are:")
for _, m := range pipeline.maintainers {
    fmt.Printf(
        "%s (address: %s)\n",
        m.full_name, m.address.text)
}
```

5.4 Serialization

Assuming the deserialized `pipeline`, you serialize it back into a JSONable `map[string]interface{}`:

```
var err error
var jsonable map[string]interface{}
jsonable, err = address.PipelineToJSONable(pipeline)
```

5.5 Implementation Details

5.5.1 Representation

Go representation of Mapry types tries to be as straight-forward as possible. The following tables show how Mapry types are mapped to Go types in generated Go code.

Table 1: Primitive types

Mapry type	Go type
Boolean	`bool`
Integer	`int64`
Float	`float64`
String	`string`
Path	`string`
Date	`time.Time`
Time	`time.Time`
Datetime	`time.Time`
Time zone	`*time.Location`
Duration	`time.Duration`

Table 2: Aggregated types (of a generic type T)

Mapry type	Go type
Array	`[]T`
Map	`map[string]T`

Table 3: Composite types

Mapry type	Go type
Reference to an instance of class T	`*T`
Embeddable structure T	`struct T`
Optional property of type T	`*T`

Table 4: Graph-specific structures

Mapry type	Go type
Registry of instances of class T	`map[string]*T`

5.5.2 Numbers

The standard encoding/json package uses double-precision floating-point numbers (`float64`) to represent both floating-point and integral numbers. Mapry-generated Go code follows this approach and assumes that all numbers are represented as `float64`. This assumption has various implications on what numbers can be represented.

The set of representable floating-point numbers equals thus that of `float64`, namely -1.7976931348623157e+308 to 1.7976931348623157e+308 with the smallest above zero being 2.2250738585072014e-308. Hence Mapry also represents floating points as `float64`.

Unlike floating-point numbers, which are simply mirroring internal and JSONable representation, Mapry represents integers as `int64` which conflicts with JSONable representation of numbers as `float64`. Namely, according to IEEE 754 standard, `float64` use 53 bits to represent digits and 11 bits for the exponent. This means that you can represent all the integers in the range [-2^53, 2^53] (2^53 == 9,007,199,254,740,992) without a loss of precision. However, as you cross 2^53, you lose precision and the set of representable integers becomes sparse. For example, 2^53 + 7 is 9,007,199,254,740,999 while it will be represented as 9,007,199,254,741,000.0 (2^53 + 8) in `float64`. Hence, you can precisely represent 2^53 + 8, but not 2^53 + 7, in your JSONable.

Unfortunately, most JSON-decoding packages (*e.g.,* encoding/json) will silently ignore this loss of precision. For example, assume you supply a string encoding a JSON object containing an integer property set to 2^53 + 7. You pass this string through `encoding/json` to obtain a JSONable and then pass it on to Mapry for further parsing. Since

Mapry does not directly operate on the string, but on an intermediate JSONable representation (which represents numbers as `float64`), your Mapry structure ends up with integer representations that diverges from the original string.

Note that this is a practical problem and not merely a theoretical one. For example, unique identifiers are often encoded as 64-bit integers. If they are generated randomly (or use 64-bits to encode extra information *etc.*) you should represent them in JSON as strings and not numbers. Otherwise, you will get an invalid unique identifier once you decode the JSON.

Furthermore, Mapry representation of integers with 64-bits restricts the range of representable integers to [-2^64, 2^64 - 1] (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). In contrast, JSONable representation uses `float64` and hence can represent the above-mentioned wider range of `float64` (-1.8e+308 to 1.8e+308). Due to this difference in representations, Mapry-generated code will raise an error if a number needs to be parsed into an integer that is out of 64-bit range.

5.5.3 Date/time Format Directives

Go standard package `time` diverges from many other languages (including C++ and Python) in that it does not support strftime/strptime directives, but a special (american-centered) date/time format of its own (see time.Format). Such format causes a couple of repercussions:

- First, fractions of seconds are not supported (akin to C/C++ `ctime` library).
- Second, certain parts of the format, while unproblematic in strftime directives, cause conflicts in Go. For example, the format `"Sun goes up:  %Y-%m-%d %H:%M:%S"` will be misinterpreted since `Sun` will be understood as abbreviated weekday in Go. Mapry detects such conflicts as soon as you try to generate Go code and raise an error. However, we leave it to the user to decide to generate code in other languages even though Go code can not be generated.

  Unfortunately, escape codes are not supported in `time` package and this problem can not be resolved.

5.5.4 Durations

Go represents durations as `time.Duration` which in fact counts the nanoseconds as `int64` (see time.Duration).

Mapry will parse the duration strings into `time.Duration`. Similar to problems in C++ generated code (see *Durations in C++*), `time.Duration` can not capture all the strings representable by ISO 8601 period strings. Number of nanoseconds are limited by the range of `int64` and can not span periods as short as 300 years (`PY300`). Furthermore, periods at finer granularity than nanoseconds are impossible to parse either (*e.g.*, `PT0.00000000004`). If you need to specify such durations, you need to specify the value as string and parse them manually.

Mind that durations in other language might introduce additional constraints. For example, Python represents durations as microseconds (see *Durations in Python*).

CHAPTER

SIX

PYTHON SPECIFICS

Mapry generates a Python module which defines structures and several de/serialization functions translating from/to JSONable `Any` values.

6.1 Settings

Mapry needs you to specify the following Python-specific settings in the schema:

`module_name` specifies the fully qualified base module name of the generated code.

For example, `book.address`.

`path_as` defines the type of the paths in the generated code.

Mapry supports: `str` and `pathlib.Path`.

`timezone_as` defines the type of the time zones in the generated code.

Mapry supports: `str` and `pytz.timezone`.

`indention` defines the indention of the generated code. Defaults to four spaces and can be omitted.

For example, `"  "` (two spaces)

6.2 Generated Code

All the files live in a single directory. The module intra-dependencies are referenced using the fully qualified base module name given as `module_name` in the schema.

Here is the overview of the generated files (in order of abstraction).

- `__init__.py` defines the general structures of the object graph (embeddable structures, classes, object graph itself *etc.*).
- `parse.py` defines general parsing structures such as parsing errors.
- `fromjsonable.py` defines parsing of the object graph from a JSONable dictionary.
- `tojsonable.py` defines serialization of the object graph to a JSONable dictionary.

The example of the generated code for the schema given in *Introductory Example* is available in the repository.

6.3 Deserialization

Given the schema provided in *Introductory Example* and assuming you obtained the JSONable value using, *e.g.*, `json` module from the standard library, you deserialize it to an object graph as follows.

```
# Obtain a JSONable
pth = '/path/to/the/file.json'
with open(pth, 'rt') as fid:
    value = json.load(fid)

# Parse the JSONable
errors = book.address.parse.Errors(cap=10)

pipeline = book.address.fromjsonable.pipeline_from(
    value=value,
    ref=pth + '#',
    errors=errors)

if not errors.empty():
    for error in errors.values():
        print("{}: {}".format(error.ref, error.message), file=sys.stderr)

    return 1
```

You can now access the object graph `pipeline`:

```
print('Maintainers are:')
for maintainer in pipeline.maintainers:
    print('{} (address: {})'.format(
        maintainer.full_name,
        maintainer.address.text))
```

6.4 Serialization

You serialize back the object graph `pipeline` into a JSONable by:

```
jsonable = book.address.tojsonable.serialize_pipeline(
    pipeline,
    ordered=True)
```

The `jsonable` can be further serialized to a string by `json.dumps(.)` from the standard library:

```
text = json.dumps(jsonable)
```

6.5 Implementation Details

6.5.1 Representation

Mapry directly maps its types to corresponding Python types. The mapping is presented in The following tables.

Table 1: Primitive types

Mapry type	Python type
Boolean	`bool`
Integer	`int`
Float	`float`
String	`str`
Path	`str` or `pathlib.Path` (depending on `path_as` setting)
Date	`datetime.date`
Time	`datetime.time`
Datetime	`datetime.datetime`
Time zone	`str` or `datetime.tzinfo` (depending on `timezone_as` setting)
Duration	`datetime.timedelta`

Table 2: Aggregated types (of a generic type T)

Mapry type	Python type
Array	`typing.List[T]`
Map	`typing.MutableMapping[str, T]`

Table 3: Composite types

Mapry type	Python type
Reference to an instance of class T	`T`
Optional property of type T	`typing.Optional[T]`

Table 4: Graph-specific structures

Mapry type	Python type
Registry of instances of class T	`typing.MutableMapping[str, T]`

6.5.2 Unordered and Ordered Mappings

When parsing a JSONable, Mapry inspects the types of the mappings to decide whether to keep or ignore the order of the keys. Namely, if the mapping is an instance of `collections.OrderedDict`, the corresponding Mapry representation will also be `collections.OrderedDict`. Analogously for an unordered mapping, if the JSONable mapping is given as `dict`, Mapry will also represent it as `dict`. This distinction is applied both to Mapry maps as well as registries of class instances.

When you serialize a Mapry structure to a JSONable, it is up to you to decide whether you want the mappings ordered or not. This is specified with the `ordered` argument. For example, consider a function generated to serialize the graph from *Introductory Example*:

```
def serialize_pipeline(
        instance: book.address.Pipeline,
        ordered: bool = False
) -> typing.MutableMapping[str, typing.Any]:
    """
    serializes an instance of Pipeline to a JSONable.

    :param instance: the instance of Pipeline to be serialized
    :param ordered:
        If set, represents the instance properties and class registries
        as a ``collections.OrderedDict``.
        Otherwise, they are represented as a ``dict``.
    :return: JSONable representation
    """
    if ordered:
        target = (
            collections.OrderedDict()
        )  # type: typing.MutableMapping[str, typing.Any]
    else:
        target = dict()

    ...

    return target
```

6.5.3 Numbers

Python 3 represents integer numbers as an unbounded `int` (see stdtypes), unlike C++ or Go (*e.g.*, see *Numbers in C++* or *Numbers in Go*, respectively) which represents numbers with bounded 64-bits integers. Since Mapry also relies on `int` to represent integers, this means that you can use unbounded integer representation in generated Python code as long as this code does not need to work with other languages.

This is particularly important when you serialize Mapry structures into JSONables. As soon as you need interoperability with, say, C++ or Go, the resulting JSONable will fail to parse. This is a limitation that Mapry does not check for at the moment. We leave it to the user of the generated code to decide how it will be used and what extra checks need to be performed since the JSONable is valid from the point-of-view of the Python 3 code.

In contrast to integers, Python 3 represents floating-point numbers (`float`) with a bounded 64-bit double-precision numbers (according to IEEE 754). This representation is also used by Mapry. This limits the range of representable numbers from -1.7976931348623157e+308 to 1.7976931348623157e+308. Note also that the closer you get to the range bounds, the sparser the representable numbers due to how floating-point numbers are represented by IEEE 754.

6.5.4 Durations

Durations are given in Python 3 as `datetime.timedelta`, a structured normalized representation of days, seconds and microseconds (see datetime.timedelta) between two instants.

The internal representation introduces the following limits:

- 0 <= microseconds < 1000000
- 0 <= seconds < 3600*24 (the number of seconds in one day)
- -999999999 <= days <= 999999999

When fraction of microseconds are specified:

> If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

Such internal timedelta structures can pose problems when you are de/serializing from JSONables coming from the code generated in other languages. You face a mismatch in granularity and range (see *Durations in C++* and *Durations in Go*).

Mapry generates C++ and Go code which uses nanoseconds to specify durations. The durations of nanosecond granularity can not be captured in Python 3 since Python 3 stores only microseconds and not nanoseconds. This can cause silent hard-to-trace truncation since Python 3 stores microseconds.

Second, `datetime.timedelta` spans a practically inexhaustible time frame which can not be fit into 64-bit integers use to represent nanoseconds in C++ and Go and can lead to overflows. For example, you can represent `PY300` without problems as duration in Mapry-generated Python 3 code, but it will overflow in C++ and Go code.

If you need to handle fine-grained and/or long durations in different languages, you better pick either a custom string or integer representation that aligns better with your particular use case.

CHAPTER

SEVEN

INSTALLATION

We provide a prepackaged PEX file that can be readily downloaded and executed. Please see the Releases section.

If you prefer to use Mapry as a library (*e.g.*, as part of your Python-based build system), install it with pip:

```
pip3 install mapry
```

CHAPTER

EIGHT

COMMAND-LINE USAGE

Mapry provides a single point-of-entry for all the code generation through `mapry-to` command.

To generate the code in different languages, invoke:

For **C++**:

```
mapry-to cpp \
    --schema /path/to/schema.json \
    --outdir /path/to/cpp/code
```

For **Go**:

```
mapry-to go \
    --schema /path/to/schema.json \
    --outdir /path/to/go/code
```

For **Python**:

```
mapry-to py \
    --schema /path/to/schema.json \
    --outdir /path/to/py/code
```

If the output directory does not exist, it will be created. Any existing files will be silently overwritten.

CHAPTER NINE

MAPRY MODULE

9.1 mapry

Serialize and deserialize object graphs from/to JSONables.

class mapry.**Array**(*values*, *minimum_size=None*, *maximum_size=None*)
Represent a type of arrays.

class mapry.**Boolean**
Represent booleans.

class mapry.**Class**(*name*, *plural*, *description*, *ref*, *id_pattern=None*)
Represent a mapry class.

class mapry.**Cpp**
List settings for the generation of the C++ code.

class mapry.**Date**(*fmt=None*)
Represent a type of dates.

class mapry.**Datetime**(*fmt=None*)
Represent a type of time points.

class mapry.**Duration**
Represent a type of durations (ISO 8601 formatted).

class mapry.**Embed**(*name*, *description*, *ref*)
Represent an embeddable structure.

class mapry.**Float**(*minimum=None*, *exclusive_minimum=False*, *maximum=None*, *exclusive_maximum=False*)
Represent a type of floating-point numbers.

class mapry.**Go**
List settings for the generation of the Go code.

class mapry.**Graph**
Represent an object graph.

class mapry.**Integer**(*minimum=None*, *exclusive_minimum=False*, *maximum=None*, *exclusive_maximum=False*)
Represent a type of integer numbers.

class mapry.**Map**(*values*)
Represent a type of mappings string -> mapry value.

class mapry.**Path**(*pattern=None*)
Represent a type of paths in the file system.

class mapry.**Property**(*ref*, *name*, *a_type*, *description*, *json*, *optional*, *composite*)
Represent a property of a composite structure.

class mapry.**Py**
List settings for the generation of the Python code.

class mapry.**Schema**(*graph*, *cpp*, *go*, *py*)
Represent a schema of an object graph.

class mapry.**String**(*pattern=None*)
Represent a type of strings.

class mapry.**Time**(*fmt=None*)
Represent a type of times of the day.

class mapry.**TimeZone**
Represent a type of time zones (according to the IANA identifier).

class mapry.**Type**
Represent a type of mapry values.

mapry.**iterate_over_types**(*graph*)
Iterate over all the value types defined in a graph.

This includes the value types of arrays and maps as well as types of properties of classes and embeddable structures.

Parameters **graph** (*Graph*) – mapry definition of the object graph

Return type Iterable[Tuple[*Type*, str]]

Returns iteration over the mapry type definitions and their reference paths

mapry.**needs_type**(*a_type*, *query*)
Search query recursively starting with a_type.

The search first checks if a_type is equal query. Otherwise, the search continues through value types, if it is an aggregated type (such as array or map) or through types of its properties, if it is a composite type (such as graph, embeddable structure or class).

Parameters

- **a_type** (*Type*) – type to inspect
- **query** (Type[*Type*]) – type to search for

Return type bool

Returns True if query found recursively in a_type

mapry.**references**(*a_type*)
Inspect recursively which classes are referenced by a_type.

Parameters **a_type** (Union[*Class*, *Embed*]) – class or embeddable structure to inspect

Return type List[*Class*]

Returns list of referenced classes

9.2 mapry.parse

Parse a mapry schema.

`mapry.parse.schema_from_json_file(path)`
Parse and validate the given JSON-encoded schema from a file.

Parameters **path** (Path) – to the JSON-encoded mapry schema.

Return type *Schema*

Returns parsed schema

`mapry.parse.schema_from_mapping(mapping, ref)`
Parse mapry schema from the given mapping.

Parameters

- **mapping** (Mapping[str, Any]) – to be parsed
- **ref** (str) – reference path to the schema

Return type *Schema*

Returns parsed schema

9.3 mapry.cpp.generate

Generate the C++ code to parse and serialize a mapry object graph.

9.3.1 mapry.cpp.generate.types_header

Generate the code that defines the types of the object graph.

`mapry.cpp.generate.types_header.generate(graph, cpp)`
Generate the header file that defines the types of the object graph.

Parameters

- **graph** (*Graph*) – definition of the object graph
- **cpp** (*Cpp*) – C++ settings

Return type str

Returns content of the header file

Ensures

- `result.endswith('\n')`

9.3.2 mapry.cpp.generate.parse_header

Generate the header of the general parsing structures.

`mapry.cpp.generate.parse_header.generate(cpp)`

Generate the header file defining the parsing structures.

Parameters **cpp** (*Cpp*) – C++ settings

Return type `str`

Returns content of the header file

Ensures

- `result.endswith('\n')`

9.3.3 mapry.cpp.generate.parse_impl

Generate the code implementing the general parse structures.

`mapry.cpp.generate.parse_impl.generate(cpp, parse_header_path)`

Generate the implementation file of the parsing structures.

Parameters

- **cpp** (*Cpp*) – C++ settings
- **parse_header_path** (`str`) – path to the header file that defines the general parsing structures

Return type `str`

Returns content of the implementation file

Ensures

- `result.endswith('\n')`

9.3.4 mapry.cpp.generate.jsoncpp_header

Generate the header for de/serialization from/to Jsoncpp values.

`mapry.cpp.generate.jsoncpp_header.generate(graph, cpp, types_header_path, parse_header_path)`

Generate the header file for de/serialization from/to Jsoncpp.

Parameters

- **graph** (*Graph*) – definition of the object graph
- **cpp** (*Cpp*) – C++ settings
- **types_header_path** (`str`) – path to the header file that defines the types of the object graph
- **parse_header_path** (`str`) – path to the header file that defines the general parsing structures

Return type `str`

Returns content of the header file

Ensures

- `result.endswith('\n')`

9.3.5 mapry.cpp.generate.jsoncpp_impl

Generate the implementation of de/serialization from/to Jsoncpp values.

`mapry.cpp.generate.jsoncpp_impl.`**`generate`**(*graph*, *cpp*, *types_header_path*, *parse_header_path*, *jsoncpp_header_path*)

Generate the implementation file for de/serialization from/to Jsoncpp.

Parameters

- **graph** (*Graph*) – definition of the object graph
- **cpp** (*Cpp*) – C++ settings
- **types_header_path** (`str`) – defines the types of the object graph
- **parse_header_path** (`str`) – defines the general parsing structures
- **jsoncpp_header_path** (`str`) – defines parsing and serializing functions from/to Jsoncpp

Return type `str`

Returns content of the implementation file

Ensures

- `result.endswith('\n')`

9.4 mapry.py.generate

Generate the Python code to parse and serialize a mapry object graph.

9.4.1 mapry.py.generate.types

9.4.2 mapry.py.generate.parse

Generate the code that defines general structures required for parsing.

`mapry.py.generate.parse.`**`generate`**(*graph*, *py*)

Generate the source file to define general structures required for parsing.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **py** (*Py*) – Python settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.4.3 mapry.py.generate.fromjsonable

9.4.4 mapry.py.generate.tojsonable

Generate the code that serializes the object graph to a JSONable.

`mapry.py.generate.tojsonable.`**`generate`**(*graph*, *py*)

Generate the source file to parse an object graph from a JSONable object.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **py** (*Py*) – Python settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5 mapry.go.generate

Generate the Go code to parse and serialize a mapry object graph.

9.5.1 mapry.go.generate.types

Generate the code that defines the types of the object graph.

`mapry.go.generate.types.`**`generate`**(*graph*, *go*)

Generate the source file that defines the types of the object graph.

Parameters

- **graph** (*Graph*) – definition of the object graph
- **go** (*Go*) – Go settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5.2 mapry.go.generate.parse

Generate the code that defines general structures required for parsing.

`mapry.go.generate.parse.`**`generate`**(*go*)

Generate the souce file to define general structures required for parsing.

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5.3 mapry.go.generate.fromjsonable

Generate the code that parses the object graph from a JSONable structure.

`mapry.go.generate.fromjsonable.`**`generate`**(*graph*, *go*)
Generate the source file to parse an object graph from a JSONable object.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **go** (*Go*) – Go settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5.4 mapry.go.generate.fromjsonable_test

Generate the code to test parsing of object graphs from JSONables.

`mapry.go.generate.fromjsonable_test.`**`generate`**(*graph*, *go*)
Generate the source file to test parsing from a JSONable object.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **go** (*Go*) – Go settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5.5 mapry.go.generate.tojsonable

Generate the code that serializes the object graph to a JSONable.

`mapry.go.generate.tojsonable.`**`generate`**(*graph*, *go*)
Generate the source file to serialize an object graph to a JSONable object.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **go** (*Go*) – Go settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

9.5.6 mapry.go.generate.tojsonable_test

Generate the code to test serializing object graphs to JSONables.

`mapry.go.generate.tojsonable_test.`**`generate`**(*graph*, *go*)

Generate the source file to test serializing to a JSONable.

Parameters

- **graph** (*Graph*) – mapry definition of the object graph
- **go** (*Go*) – Go settings

Return type `str`

Returns content of the source file

Ensures

- `result.endswith('\n')`

CHAPTER TEN

RELATED PROJECTS

We give here a non-comprehensive list of related de/serialization projects. We indicate how they differ from Mapry and explain why we took pains to develop (and maintain!) our own tool instead of using an existing one.

- Standard JSON libraries all support object trees, but not object graphs. Moreover, they do not support data based on a schema. While this is handy when the structure of your data is unknown at runtime, it makes code unnecessarily more difficult to maintain when the structure is indeed known in advance.
- There is a large ecosystem around structured objects and their serialization based on property annotations (*e.g.*, Rapidschema (C++), encoding/json (Go) or Jackson (Java)). While some of them support handling object graphs (usually through custom logic), we found the lack of polyglot support (and resulting maintenance effort required by synchronization of custom de/serialization rules across languages) a high barrier-to-usage.
- Standard or widely used serialization libraries such as Boost.Serialization (C++), Gob (Go) or Pickle (Python) serialize object graphs out-of-the-box and handle impedance mismatch well. However, the representation of the serialized data is barely human-readable and difficult to get right in a polyglot setting due to a lack of common poly-language libraries (*e.g.*, reading pickled data structures in C++). We deemed it a Herculean task to maintain the corresponding de/serializations accross different languages.
- Popular serializers such as Protocol Buffers or Cap'n Proto support only object trees. If you need to work with cross-references in the serialized message, you need to dereference instances yourself. We found manual dereferencing in code to be error prone and lead to a substantial code bloat.
- Flatbuffers handle object graphs natively, but exhibit a great deal of impedance mismatch through lack of maps and sophisticated data types such as date/time, duration *etc.*
- Language-specific serializers such as ThorSerializer (C++), JavaScript Object Graph (Javascript), Serializr (Javascript) and Flexjson (Java) serialize object graphs with satisfying, but varying degree of structure enforcement and readability. Most approaches require the developer to annotate the structures with decorators which the libraries use to parse and serialize data. As long as you use a single-language setting and care about the data being readable, these solutions work well. However, it is not clear how they can be adapted to a multi-language setting where system components written in different languages need to inter-operate.
- JSON for Linking Data and JSON Graph are conventions to provide a systematic approach to modeling the object graphs in JSON. While these conventions look promising, we found the existing libraries lacking for production-ready code. On a marginal note, the JSON representations seem unnecessarily verbose when representing references.
- JVM serializers presents a report on different object serializers running on top of Java Virtual Machine. The serializers are evaluated based on their run time and size.

CHAPTER ELEVEN

FUTURE WORK

While Mapry satisfies very well many of our practical needs, there are countless possible improvement vectors. If you feel strong about any of the listed improvements (or you have another one in mind), please create an issue and help us discuss it.

New primitive types. We tried to devise a practical set of primitive types that covers most use cases. However, we do not know our (existing or potential) user base and our assumptions on what is necessary might be wrong.

New aggergated types. So far, we introduced only arrays and maps as aggregated types since they are JSON-native.

While JSON does not support aggregated types such as sets, the sets are at the core of many data models and would definitely merit a representation in Mapry. Please let us know your opinion about what would be a conventional way of representing sets in JSON.

Elaborate composite type system. We limited the composite type system to a graph, classes and embeddable structures for simplicity following Go's approach (lack of inheritance, tuples and unions by design). We find that optional properties cover most of the use cases which are also covered by inheritance, tuples or unions. Hence, we thought that having optional properties is enough and did not want to complicate further the type system.

Please feel free to convince us of the contrary and tell us how inheritance, tuples or unions should be handled. In particular, we do not really know what would be a conventional way of dealing with such a type system in Go.

Moreover, it is not clear to us how to deal with variance in aggregated types (covariance, contravariance or invariance) since different languages follow different approaches. For example, C++ `std::list` and Python's `List` are invariant (*e.g.,* `std::list<Employee>` can not pass as `std::list<Person>`), while Python `Sequence` is covariant (at least in mypy generics). Admittedly, we are a bit lost how to approach this issue and are open to suggestions.

Better data constraints. We are convinced that constraints (such as min/max ranges) make data structures more maintainable and prevent many of the errors early. However, Mapry's current constraints are quite limited and probably need extensions. Please let us know which constraints you would need and how you would like to specify them.

Unfortunately, we can support only the most basic constraints. We do not have the time resources to include a declarative or imperative constraint language that would automatically compile into the generated code. Notwithstanding the lack of time, we strongly believe that such a language would be beneficial and are open for cooperation if you think you could help us tackle that challenge.

Efficiency of de/serialization. Mapry was optimized for readability of generated code rather than the efficiency of de/serialization. Multiple improvements are possible here.

Obviously, the generated de/serialization code could be optimized while still maintaining the readability. Please let us know which practical bottlenecks you experienced so that we know where/how to focus our optimization efforts.

Since Mapry does not depend on the source of the JSONable data, you can already use faster JSON-parsing libraries (*e.g.*, fastjson (Go) or orjson (Python)). However, in C++ setting where no standard JSONable structure exists, we could introduce an additional code generator based on faster JSON-parsing libraries such as rapidjson.

Fast de/serialization of character streams. Instead of operating on JSONable structures which are wasteful of memory and computational resources, we could generate de/serialization code that operates on streams of characters. Since schema is known, we could exploit that knowledge to make code work in one pass, be frugal in memory (*e.g.*, consume only as much memory as is necessary to hold the object graph) and be extremely fast (since the data types are known in advance).

Additionally, when the language is slow (*e.g.*, Python), the code can be made even faster by generating it in the most efficient language (*e.g.*, C) together with a wrapper in the original language.

For an example of such an approach based on schema knowledge, see easyjson (Go).

Improve readability of generated code. While we find the generated code readable, the readability lies in the eye of the beholder. Please let us know which spots were hard for you to parse and how we could improve them.

Runtime checks at serialization. We designed Mapry to perform runtime validation checks only at deserialization since we envisioned its main input to be generated by humans. However, if you construct an object graph programmatically, you need to serialize it and then deserialize it in order to validate the contracts. While this works in cases with small data, it would be computationally wasteful on large object graphs.

We are thinking about introducing validation at serialization as well (triggered by a dedicated flag argument). Please let us know if you miss this functionality and what would you like to have covered.

CHAPTER TWELVE

CONTRIBUTING

We are very grateful for and welcome contributions: be it opening of the issues, discussing future features or submitting pull requests.

To submit a pull request:

- Check out the repository.
- In the repository root, create the virtual environment:

```
python3 -m venv venv3
```

- Activate the virtual environment:

```
source venv3/bin/activate
```

- Install the development dependencies:

```
pip3 install -e .[dev]
```

- Implement your changes.
- Run *precommit.py* to execute pre-commit checks locally.

12.1 Live tests

We also provide live tests that generate, compile and run the de/serialization code on a series of tests cases. These live tests depend on build tools of the respective languages (*e.g.*, gcc and CMake for C++ and go compiler for Go, respectively).

You need to install manually the build tools. Afterwards, create a separate virtual environment for the respective language and install Python dependencies for the respective language (*e.g.*, Conan in case of C++) given as `test*` requirements in setup.py.

The workflow for C++ looks as follows:

```
# Create a separate virtual environment
python3 -m venv venv-cpp

# Activate it
. venv-cpp/bin/activate

# Install the dependencies of C++ live tests
pip3 install -e .[testcpp]
```

(continues on next page)

(continued from previous page)

```
# Run the live tests
./tests/cpp/live_test_generate_jsoncpp.py
```

For Go:

```
python3 -m venv venv-go
. venv-go/bin/activate
pip3 install -e .[testgo]
./tests/go/live_test_generate_jsonable.py
```

For Python:

```
python3 -m venv venv-py
. venv-py/bin/activate
pip3 install -e .[testpy]./p
./tests/py/live_test_generate_jsonable.py
```

CHAPTER

THIRTEEN

VERSIONING

We follow Semantic Versioning. We extended the standard semantic versioning with an additional format version. The version W.X.Y.Z indicates:

- W is the format version (data representation is backward-incompatible),
- X is the major version (library interface is backward-incompatible),
- Y is the minor version (library interface is extended, but backward-compatible), and
- Z is the patch version (backward-compatible bug fix).

CHAPTER FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

mapry, 37
mapry.cpp.generate, 39
mapry.cpp.generate.jsoncpp_header, 40
mapry.cpp.generate.jsoncpp_impl, 41
mapry.cpp.generate.parse_header, 40
mapry.cpp.generate.parse_impl, 40
mapry.cpp.generate.types_header, 39
mapry.go.generate, 42
mapry.go.generate.fromjsonable, 43
mapry.go.generate.fromjsonable_test, 43
mapry.go.generate.parse, 42
mapry.go.generate.tojsonable, 43
mapry.go.generate.tojsonable_test, 44
mapry.go.generate.types, 42
mapry.parse, 39
mapry.py.generate, 41
mapry.py.generate.parse, 41
mapry.py.generate.tojsonable, 42

INDEX

A

Array (*class in mapry*), 37

B

Boolean (*class in mapry*), 37

C

Class (*class in mapry*), 37
Cpp (*class in mapry*), 37

D

Date (*class in mapry*), 37
Datetime (*class in mapry*), 37
Duration (*class in mapry*), 37

E

Embed (*class in mapry*), 37

F

Float (*class in mapry*), 37

G

generate() (*in module mapry.cpp.generate.jsoncpp_header*), 40
generate() (*in module mapry.cpp.generate.jsoncpp_impl*), 41
generate() (*in module mapry.cpp.generate.parse_header*), 40
generate() (*in module mapry.cpp.generate.parse_impl*), 40
generate() (*in module mapry.cpp.generate.types_header*), 39
generate() (*in module mapry.go.generate.fromjsonable*), 43
generate() (*in module mapry.go.generate.fromjsonable_test*), 43
generate() (*in module mapry.go.generate.parse*), 42
generate() (*in module mapry.go.generate.tojsonable*), 43
generate() (*in module mapry.go.generate.tojsonable_test*), 44
generate() (*in module mapry.go.generate.types*), 42
generate() (*in module mapry.py.generate.parse*), 41
generate() (*in module mapry.py.generate.tojsonable*), 42
Go (*class in mapry*), 37
Graph (*class in mapry*), 37

I

Integer (*class in mapry*), 37
iterate_over_types() (*in module mapry*), 38

M

Map (*class in mapry*), 37
mapry
    module, 37
mapry.cpp.generate
    module, 39
mapry.cpp.generate.jsoncpp_header
    module, 40
mapry.cpp.generate.jsoncpp_impl
    module, 41
mapry.cpp.generate.parse_header
    module, 40
mapry.cpp.generate.parse_impl
    module, 40
mapry.cpp.generate.types_header
    module, 39
mapry.go.generate
    module, 42
mapry.go.generate.fromjsonable
    module, 43
mapry.go.generate.fromjsonable_test
    module, 43
mapry.go.generate.parse
    module, 42
mapry.go.generate.tojsonable
    module, 43
mapry.go.generate.tojsonable_test
    module, 44
mapry.go.generate.types
    module, 42
mapry.parse

module, 39
mapry.py.generate
module, 41
mapry.py.generate.parse
module, 41
mapry.py.generate.tojsonable
module, 42
module
mapry, 37
mapry.cpp.generate, 39
mapry.cpp.generate.jsoncpp_header, 40
mapry.cpp.generate.jsoncpp_impl, 41
mapry.cpp.generate.parse_header, 40
mapry.cpp.generate.parse_impl, 40
mapry.cpp.generate.types_header, 39
mapry.go.generate, 42
mapry.go.generate.fromjsonable, 43
mapry.go.generate.fromjsonable_test, 43
mapry.go.generate.parse, 42
mapry.go.generate.tojsonable, 43
mapry.go.generate.tojsonable_test, 44
mapry.go.generate.types, 42
mapry.parse, 39
mapry.py.generate, 41
mapry.py.generate.parse, 41
mapry.py.generate.tojsonable, 42

N

needs_type() (*in module mapry*), 38

P

Path (*class in mapry*), 37
Property (*class in mapry*), 37
Py (*class in mapry*), 38

R

references() (*in module mapry*), 38

S

Schema (*class in mapry*), 38
schema_from_json_file() (*in module mapry.parse*), 39
schema_from_mapping() (*in module mapry.parse*), 39
String (*class in mapry*), 38

T

Time (*class in mapry*), 38
TimeZone (*class in mapry*), 38
Type (*class in mapry*), 38