KTH Computer Science
and Communication

Comparison between JSON and YAML for data serialization

MALIN ERIKSSON
VICTOR HALLBERG

Bachelor's Thesis in Computer Science (6 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011

Supervisor: Mads Dam
Examiner: Mads Dam

malier@kth.se
victorha@kth.se

Abstract

This report determines and discusses the primary differences between two different serialization formats, namely YAML and JSON. A general introduction to the concepts of serialization and parsing is provided first, which also explains how they can be used to transfer and store data. This is followed by an analysis of the YAML and JSON formats, where functionality, primary use cases, and syntax is described. In addition to this the percieved performance of implementations for both formats will also be investigated by conducting a number of tests. Using the combined background information and results from the tests, conclusions regarding the main differences between the two are then determined and discussed.

Referat

Denna rapport tar upp och diskuterar primära skillnader mellan två olika serialiseringsformat; YAML och JSON. Först ges en övergripande introduktion till begreppen serialisering och parsing, som även förklarar hur de kan användas för att överföra och lagra data. Därefter följer en mer djupgående analys av YAML och JSON, där funktionalitet, primära användningsområden samt syntax beskrivs. Utöver detta undersöks även prestandan hos implementationer av de olika formaten med hjälp av ett antal tester. Slutligen används den samlade bakgrundsinformationen och resultaten från de genomförde testerna för att påvisa de största skillnaderna mellan dem.

Contents

Preface

The work of this study was divided into different stages. It started with a pre-study, which was conducted by both group members. Some theoretical research on the subject was then collected, which Malin was primarily responsible of. This was done in parallel with preparations for the performance testing being done by Victor. Throughout the whole process, both of us have been writing different parts of the report. In general, most of the technical issues have been Victors responsibilities, whereas Malin has been doing background research, and compiling found facts. Most sections has finally been revised by both members.

Chapter 1
Introduction

This paper discusses and compares different serialization formats in computer science. In addition to a background of the serialization technique and how it is utilized as a method today, it also compares two important light-weight data interchange formats, namely JSON and YAML. Although they are quite similar when it comes to usability, there are some distinctions between them, mostly regarding design choices and syntax. These choices do however affect their scope of use. The report aims to discuss the various differences between the languages, along with the resulting consequences regarding performance and usability in different use cases which they lead to.

1.1 Problem statement

There has been an increase in discussions comparing the usability of YAML and JSON as serialization formats in recent years. Even though there are multiple thoughts and opinions on the net, there is a lack of actual general investigation on the subject.

The primary aim for this project is to determine and compare the major differences between YAML and JSON from multiple perspectives. This will not only be done through a performance test, the comparison is also based on collected facts and research. From this, conclusions can be drawn regarding their usability and different scope of use. This comparison will help define the gap between them (if such exists), and will hopefully provide some guidelines to consider for future development involving data serialization.

Chapter 2
Background

This chapter explains and defines background facts regarding the different parts of this report. It includes information about the serialization and parsing process, but focuses on the serialization languages to be compared - YAML and JSON.

2.1 Serialization

2.1.1 Definition

Serialization is a process for converting a data structure or object into a format that can be transmitted through a wire, or stored somewhere for later use [8].

In terms of serialization there are a legion of different ways and formats that can be used. Which method and format to choose depends on the requirements set up on the object or data, and the use for the serialization (sending or storing). The choice may also affect the size of the serialized data as well as serialization/deserialization performance in terms of processing time and memory usage.

2.1.2 General method

Common for all serialization methods is the procedure of reading data as a series, once started the whole object will usually be serialized/deserialized. This enables the use of simple I/O interfaces to hold and pass on the state of an object, although difficulties arise in applications which require higher performance by having a non-linear storage organization, or when the object contains large amounts of data. These cases requires more effort to deal with, and will not be covered in this paper. The most commonly used data structures when encoding data like this are scalars, maps and sequences (lists or arrays).

Serialization is supported by many of the popular object-oriented programming languages like PHP, Ruby, Java, Smalltalk and Python along with the .NET Framework. All of these languages provide serialization methods either as implementable interface or as syntactic sugar. For example .NET provides a serializable attribute [10], and Java uses an interface named Serializable for classes to implement [9]. In Ruby, the term used for serialization is marshaling, and the language provides a module called Marshal for this [16]. This module can often be used without any changes to the definitions of the objects to be serialized. A serialization strategy can be defined in cases when you want to restrict the serialization process (all instance variables are serialized by default) or handle data in specific ways.

Most of the standard serialization implementations converts the data into a binary string, which means that the data will not easily be inspected by a human

in its serialized form. Rubys Marshal module returns a plain text string, which however is not completely readable as it contains special byte sequences and is not formatted in a way to be easily read by a human.

Example

A concrete example where serialization is needed is when storing information from an address book, in this case written in Java. Every instance contains a person with details about their address and phone number. One wants to store all instances on a server in exactly the way they are created and there are a few possible solutions;

1. By using Java serialization, which is part of the language. This can easily be done, but problems arise if the data would have to be accessible to applications written in C++, Python or another language as the data is serialized in a way unique to Java.

2. By using an improvised way of encoding the data into single strings, such as encoding four integers into for example `12:3:-23:67`. This solution requires some custom parsing code to be written, and is most efficiently used when converting very simple data.

3. By serializing the data into XML. It is an attractive method due to the fact that XML is human readable and have bindings (API libraries) for many languages, although it is space intensive and can cause performance penalties on applications.

Due to the ineffectiveness regarding these approaches mentioned above, other solutions are often desirable.

2.1.3 Scope of use

Serialization is often used when transmitting data, as has been mentioned above. Some example of such cases are when storing user preferences in an object or for maintaining security information across pages and applications. In general, when transferring objects in applications, domains, or through firewalls, serialization can be very helpful.

2.2 Parsing

2.2.1 General

The term parsing in computer science means in general to analyze written text, determining its grammatical structure from a known formal grammar. In linguistic terms, parse means analyzing and describe the grammar of a sentence. The parser splits up an expression into tokens which are then inserted into some kind of data

structure. This data is the evaluated to interpret the meaning of each expression by the rules from given grammar, followed by execution of the appropriate action.

2.2.2 Serialization and parsing

Serialization is mainly a method to maintain easy ways of storing, in the sense of converting data and then restore it into a semantically equivalent clone. Unless the serialization method used serializes the data in a coherent order (never changing) and expects the data to be read in the same order when deserializing, parsing will have to be done when the data is to be deserialized. When deserializing, parsing is done to identify the data identifiers (attribute names or the like) and their corresponding values (while at the same time often having to discern the type of data).

The following sections aims to introduce JSON and YAML and makes no statements about differences between them. This will be discussed later on in the results and conclusions chapters.

2.3 JSON

2.3.1 General

JSON is a subset of the open ECMAScript standard [3] (which the JavaScript programming language is an implementation of). It was created to be used as a way to parse human-readable (in plain text format) representations of data into valid ECMAScript objects [7]. It is completely language independent and uses notations similar to common programming languages such as C, C++, Java, etc.

The format has grown to be very popular in cases where serialization and interchange of structured data over networks [13] and is often associated with the modern web due to the fact that it is frequently used when communication between a web server and client side web application is requested.

2.3.2 Origin

JSON was originally introduced as a written specification by Douglas Crockford in 2001 [4], who used the format within his company State Software. Crockford was not the first person to invent the object notation as other individuals had discovered it independently at about the same time, but he was the first one to give it a complete specification based on parts of the JavaScript standard. Following that he launched the JSON.org website in 2002, which still exists and currently provides a listing of JSON libraries for different programming languages [3]. It quickly grew in popularity partly thanks to its simplicity, which made it much more light weight (resulting in faster load times over the Internet) compared to XML, a format frequently used on the web. The other reason for the growth in usage is the increased use of JavaScript on the web.

JSON documents can be parsed in JavaScript by calling the built-in `eval` function with the JSON string provided as an argument. The JavaScript interpreter will then execute the parameter as JavaScript code, constructing an object with the properties defined by the JSON string. This will work due to the fact that JavaScript is a superset of JSON. Using the eval function is theoretically the most efficient way to parse JSON as it will just invoke the JavaScript interpreter (without any security/constraint cheks). This method can be said to be quite inelegant since the interpreter does not prevent any JavaScript code from being executed. In most cases a dedicated JSON parser should be used to avoid security issues and only allow valid JSON as input. Most of the modern browsers have had fast native JSON parsers since 2009, which are preferred to using `eval`[13].

2.3.3 Functionality

JSON is human-readable language, foremost designed for its simplicity and universality. It implements basic data types available to most modern programming languages[6]. The fact that it also is easy to read and parse contributes to its usefulness in programming. JSON also is language-independent, meaning that the specification is not tied to any specific programming language (it was originally based on the JavaScript object notation however). The design incorporates data types common across most modern languages.

The JSON standard does not support object references, which affects the ability to store cyclic structures for example. This functionality can be provided by an extension like `dojox.json.ref` from the Dojo Toolkit[19], enabling JSON objects to be marked with specific ids which can later be referenced to.

Complex structures can also be built as associative arrays, objects within objects. JSON objects can contain any valid data type, enabling deep data hierarchies in JSON documents.

The JSON format specification does not include support for validations of values or structure, but a external specification called JSON Schema exists as a draft[20]. JSON Schema can be used to define the structure of a JSON document much like an XML Schema, for example which data types values should have and if they are optional or required to be present. The defined schema can then be used to validate JSON documents or as a way to document application APIs.

Valid data types are[2]:

- Numbers (floating-point numbers in scientific notation, infinity is not permitted)
- Strings (with Unicode support)
- Boolean (true/false)
- Objects (associative arrays / objects with key-value pairs)

- Arrays (ordered lists)
- Null

2.3.4 Syntax

As is described above, JSON consists of objects, arrays and scalars. General syntax will be described in this section, which is intended to give an overview over the language and its usability regarding semantics. An example of a arbitrary JSON document can be found in table 2.1 (with code converted from the YAML example[6]).

- Comments are not allowed in the current standard (they were removed by the author in a later revision of the specification[4]).
- Objects (unordered collection of name/value pairs) are denoted with braces (`{}`).
- Identifiers must be enclosed in quotes (as a string) and are followed by a colon and value.
- Objects (associative arrays / objects with key-value pairs)
- Multiple key-value pairs are separated with a comma.
- Arrays (ordered set of values) are placed within brackets (`[]`) and separated by commas.
- The root node of a JSON document must be an object or an array.

2.3.5 Scope of use

JSON, considered to be a more user-friendly alternative to XML, is often used as a substitute to it. When XML has been said to contribute with a lot of unnecessary baggage, JSON documents can contain the same information while also being much more light weight and easy to read[5]. JSON is most commonly used when exchanging or storing structured data. It is especially common in Ajax web applications, where it provides a standardized data exchange format for JavaScript implementations[11].

2.3.6 Process

JSON is parsed (deserialized) in a simple character by character reading, constructing structures and object in one single pass. JavaScript implementations allows a parameter for an external function (called a reviver) to be provided, allowing more specific transformation of data. Serialization is also done in one single iteration

Table 2.1: Log file for an arbitrary application in JSON format

```
[{
  "User": "ed",
  "Time": "2001-11-23 15:01:42 -5",
  "Warning": "This is an error message for the log file"
}, {
  "User": "ed",
  "Time": "2001-11-23 15:02:31 -5",
  "Warning": "A slightly different error message."
}, {
  "User": "ed",
  "Date": "2001-11-23 15:03:17 -5",
  "Fatal": "Unknown variable \"bar\"",
  "Stack": [
    {
     "code": "x = MoreObject(\"345\\n\")\n",
      "line": 23,
      "file": "TopClass.py"
    }, {
      "code": "foo = bar",
      "line": 58,
      "file": "MoreClass.py"
    }
  ]
}]
```

A JSON document with an array containing multiple log entries.

through the data structure, where most implementations call a `to_json` (or similarly named) method, either earlier defined by the implementation or by the user, and then appends the result of this method call to the JSON output.

2.4 YAML

2.4.1 General

YAML is a recursive acronym for YAML Ain't Markup Language, emphasizing on it's design as a data storage format. It is a light-weight human readable serializing language primarily designed to be easy to read and edit. By adding a simple typing system and aliasing mechanism upon the three most common data structures used when serializing (hashes, arrays and strings) it forms a language which is easy to use, while still including more complex features[6].

2.4.2 Origin

YAML was first proposed by Clark Evans in 2001, who then designed it together with Ingy döt Net and Oren Ben Kiki[6]. The format was developed from experience

and discussions among sml-dev members on the Internet, and is still updated based on user input from the YAML-core mailing list [6].

2.4.3 Functionality

Due to the fact that YAML needs to be easily extensible and readable by humans it is mainly integrated and built upon concepts described by C, Java, Perl, Python and Ruby. The main design goals are to mimic native data structures of modern languages as much as possible, support parsing and have a consistent model to support generic tools. This can lead to some complications when generating and parsing YAML documents [6].

YAML can be considered to be a superset of JSON, providing syntax for improved human readability along with a more complete information model (support for additional types) [6]. JSON files are often valid YAML files because of the fact that JSON's semantic structure is equivalent to YAML's in line writing style (which was added in the new v1.2 specification of YAML). This means that YAML parsers adhering to the new specification also should be able to parse most JSON files.

In addition to the basic data types available in JSON, YAML also supports relational trees. Relational trees are a language construct with which references to other nodes in the YAML document can be made [6]. A node in the YAML document tree can be defined as an anchor, and later references to that anchor will then include the data of the anchored node into the node. Smart use of this feature can lead to increased readability, compactness and clarity along with less chance of data entry errors.

The YAML specification also allows user-defined data types to be declared, as well as explicit data typing. This is especially useful for serialization purposes, allowing a parser to automatically construct an object of the correct class when deserializing, instead of an generic collection.

YAML structures includes nodes and tags. A node represents a single native data structure, which can be a scalar, sequence or mapping. Each node can be marked with a tag, which restricts the set of possible values upon that node. A tag works as a identifier for data structures. YAML defines two different types of tags. Local tags are specific to a single application and start with an exclamation mark. Global tags are global across all applications and are defined as URIs [22]. Tags are primarily used to associate meta data to nodes, for example by telling the YAML parser what kind of user defined object a node represents (which the parser then may deserialize the data into).

The basic data types are [6]:

- Numbers (hexadecimal/octal, integers, floating-point numbers)
- Strings (with Unicode support)
- Boolean (true/false)

- Dates and timestamps
- Maps (associative arrays / objects with key-value pairs)
- Sequences (arrays, ordered lists)
- Null

2.4.4 Syntax

Describes YAML language and syntax, and gives an overview over the language. An example of a arbitrary YAML document can be found in table 2.2 [6].

General:

- Comments begin with a hash/number sign (`#`) and continues to the end of the current line.
- Document data hierarchy is determined by indentation using double space characters (tab characters are not allowed as indentation).

Mappings (associative arrays):

- One mapping per line, marked with an identifier followed by a colon and space (key: value).
- An inline format which mimics the JSON object notation is also available. Associative arrays are in this case enclosed in braces with items being comma separated).

Sequences (arrays):

- One item per line, marked with a dash and space.
- An alternative inline syntax exists, where the list is enclosed in brackets and items are separated by a comma followed by space.

Structures:

- Three repeated dashes denote the start of a document, and is also used to separate multiple documents in a single transmission.
- The root node of a document can be any valid data type.
- Ending a transmission along with the current document is done with three repeated dots.
- Repeating nodes are defined with an ampersand and later referenced with an asterisk, where character is followed by an identifier.

- A question mark and space in the beginning of a line denotes sets which are unordered.
- Values of user-defined data types can be denoted by prefixing them with a exclamation mark followed by the data type name, a space and finally the value.
- Explicit data type casting is done by prefixing the value in the same way as with user defined types but with an additional exclamation mark.

Strings:

- Quoting is often not required but can be, using either single or double quotes.
- The single quoted style is useful when no escaping is needed, while the double quoted style allows for escape sequences. It can span multiple lines and newlines are folded and included by a newline escape character (`\\n`)
- Strings can be written using either the standard inline style (with or without quotes) or with block notation where a initial symbol determines how newlines in the document should be handled.
- Strings can be written using either the standard inline style (with or without quotes) or with block notation where a initial symbol determines how newlines in the document should be handled.
- Strings in block notation denominated with a pipe (`|`) will have their newlines preserved, while the greater than sign (`>`) will tell the YAML parser to convert newlines to spaces.

2.4.5 Scope of use

Examples of common use cases for YAML are configuration files and log files (as seen above). It can also be used for inter-process messaging or cross language data sharing in applications. Debugging complex data structures can be simplified by using YAML to format the data, which is taken advantage of in the Ruby on Rails web framework.

2.4.6 Process

The YAML specification outlines four stages of data when loading and dumping to and from the format[6]. Native data (in the program environment) is seen as the first stage. The serialized YAML document (string) is the last stage of data. The two stages in between can be seen as working stages, where the data has been transformed into a node graph or event tree to be further processed.

Serialization, or dumping as it is referred to, is done in three distinct stages which converts the data from a native data structure into series of bytes (strings). First, a

Table 2.2: Log file for an arbitrary application in YAML format

```
---
Time: 2001-11-23 15:01:42 -5
User: ed
Warning:
 This is an error message for the log file.
---
Time: 2001-11-23 15:02:31 -5
User: ed
Warning:
 A slightly different error message.
---
Date: 2001-11-23 15:03:17 -5
User: ed
Fatal:
 Unknown variable "bar".
Stack:
- file: TopClass.py
  line: 23
  code: |
    x = MoreObject("345\n")
- file: MoreClass.py
  line: 58
  code: |-
    foo = bar
```

A YAML transmission with multiple log entries, each sent as a individual document.

directed graph is generated containing the structure - with nodes, sequences, mappings and scalars. The graph is then serialized, where sequential access mediums must be represented as ordered trees. In YAML they are created by ordered mappings, also called serialization trees. General mapping keys are unordered. Finally, the serialized tree is converted into a Unicode string.

The load (deserialization) process is also compromised of three stages, which together does the reverse. The input (a string) is parsed to create a serialization tree in which the node hierarchy, keys, values and ordering is defined. This tree is then traversed node-to-node, where the data types of values are determined and converted to, as well as constructing relations and sequences. The final step converts the representation graph in to native data structures.

Chapter 3
Methods

This chapter describes the methods used during the research and testing. This paper consists of one theoretical study, which builds conclusions from earlier research. From those conclusions a test environment is constructed for testing and performance comparison.

3.1 Testing tools and environment

3.1.1 Programming language

The tests will be conducted with programs written in the Ruby programming language. Ruby is an object-oriented language which focuses on simplicity and productivity, designed to be used for scripting as well as bigger projects[15]. It has gained popularity as a powerful scripting language, but is sometimes said to not be a high-performance language[15]. There are numerous libraries and other tools written in the language, and many are available as gems - packages which can be installed and made available to the Ruby environment using a tool called RubyGems.

3.1.2 JSON implementation

JSON serialization and deserialization will be tested using an implementation of the specification, provided by the ruby gem called json[17]. This implementation is partly written in C and uses a custom parser and Unicode conversion functions. There is also an alternative implementation called json-pure written in pure Ruby, and therefore not having any external dependencies. Json-pure is slightly slower, and we chose to use the previously mentioned implementation since it is the recommended version[17].

The json gem defines a JSON class and adds two methods to all basic types (`Object.to_json` and `Object.json_create`) when included into the Ruby environment. The JSON class provides methods for deserialization of JSON documents into Ruby data structures (`JSON.parse`), and serialization of Ruby data structures into JSON documents (`JSON.generate`)[17]. `JSON.generate` returns a compact JSON string without indendation. An alternative, `JSON.pretty_generate`, which will return a string with indendation where appropriate is also available. Both of these methods will be included in the tests.

The `to_json` method must be manually defined for custom classes to be able to serialize them, and the implementation does not support deserialization directly into the original class (due to the lack of tag/type declaration in the JSON format). Basic data types such as scalars, arrays and hashes are handled well however.

3.1.3 YAML implementation

Support for (de)serialization to and from YAML is included in the Ruby standard library (In the YAML module). This implementation follows the 1.0 version of the specification[18] but does include support for the alternative JSON syntax for sequences and mappings.

The YAML module adds two main methods for transformations. `YAML.load` can be used to parse a string or file stream containing a YAML document, while `YAML.dump` is used to serialize an object. The implementation can by default serialize most objects without having to define a custom `to_yaml` method, and outputs the class name of the object in the tag part of the YAML output. The result of this is that the YAML module can recreate a semantically identical copy of the original object when later loading the serialized data. This can be a big advantage when the data to serialize contains custom objects compared to the json gem, does not support custom classes out of the box.

3.1.4 Environment

The tests where conducted in the following computer environment.

OS	Windows 7 x64
Processor	Core 2 Duo P7350
Memory	4GB DDR3 RAM
Ruby	v1.8.7 (2010-12-23 patchlevel 330) (i386-mingw32)
JSON	json gem (ext) v1.5.1 (x86-mingw32)
YAML	version bundled with Ruby

3.2 Testing procedure

The perceived performance of JSON and YAML will be determined from a custom benchmarking script written in the Ruby programming language[14]. The script generates two different data sets (described below), which are then serialized using different Ruby implementations for both formats, followed by a deserialization run on the serialized data back into the Ruby environment. Both (de)serialization processes are done while the time taken is measured. The benchmark also logs the final file size of the serialized output. Source code for the benchmark is available at github[1].

3.3 Test data

Performance testing will be conducted with two separate sets of generated data. The first data set mimics large amounts of simple data (a single array of objects). A complex data set will also be used to test performance of the implementations for documents with deeper hierarchies.

3.3.1 Simple data set

The first data set consists of a large array (ten thousand indexes) with one-dimensional associative arrays (objects), having predefined keys and randomly generated values (an integer, a short string and a date object). This type of data set could in practice be a dictionary or the access log of a system.

3.3.2 Complex data set

The complex data set includes nested objects and additional key-value-pairs for each object. The data is generated by a custom script which constructs an hierarchy of hashes. Each hash has an integer, a short and longer string, two date objects and a child array with zero or more children of the same structure (each child possibly having a similar child array of their own). A real world example of a data set designed like this could be a page hierarchy for a website.

3.4 Tests

3.4.1 Serialization performance

The performance of the serialization process will be measured as the time taken for each serialization implementation to serialize a previously generated set of data present in the script environment into their respective formats. The execution times for each implementation, measured in seconds, will be used to determine the perceived performance.

3.4.2 Deserialization performance

Each serialization implementation will have its performance tested by using the resulting output string of the serialization process as input for the deserialization method. The measured execution time (in seconds) of the process will also be used here to determine performance.

3.4.3 Serialization output size

The final attribute to investigate is the size of the serialized data. This will be determined by investigating the size (length) of the output string from the serialization method, and will be measured in kilobytes.

Chapter 4
Results

This chapter presents the results, taken from either testing or conclusions based on the background information shown in the Background chapter.

4.1 General differences

4.1.1 Data types

See table 4.1 for a comparison of available basic data types in JSON and YAML. JSON and YAML both incorporates the most basic data types. YAML does however include better support for numbers along with a native timestamp type. It also provides functionality to explicitly tag values as specific data types (user-defined or native), which enables the parser to create a object of the correct type automatically. This feature does not exist in the JSON specification.

Table 4.1: Basic data types

Type	JSON	YAML
Integers	Yes	Yes
Floats	Scientific notation	Scientific notation
Number specifics	Not infinity	Also octal/Hexadecimal
Strings	Yes (Unicode)	Yes (Unicode)
Booleans	Yes	Yes
Arrays	Yes (sequences)	Yes
Associative arrays	Yes (objects)	Yes (mappings)
Null	Yes	Yes
Timestamps	As strings	Yes

Avaibility of basic data types in both formats (naming in parentheses).

4.1.2 Structures

Both formats supports lists and associative arrays. YAML includes functionality like object references and relational trees natively, whereas JSON doesn't. Object references can be added to JSON through third-party extensions however. Generating JSON from data which includes cyclic structures should be avoided as the format doesn't support references and thus will end up in an infinite loop unless special care is taken to prevent this.

4.1.3 Implementation

The simplicity of the specification makes parsing and generation of JSON trivial. The YAML specification explicitly outlines three stages for the parsing process. Along with the added features compared to JSON, this greatly increases the complexity of the parser and serializer.

4.1.4 Readability

JSON has a simple and very easy to learn syntax, with readable output as long as it includes whitespace in the form of indendation and line breaks. YAML, being designed to produce easily read documents, presents a cleaner output which is easier to read than an equivalent JSON document however. It has a much higher readability, while also being more compact (unless the JSON is generated without indendation). Comment syntax is available for YAML, along with multiple ways to write strings. JSON previously allowed a JavaScript-like comment syntax, but it was removed from the specification in a rather early stage.

Strings in YAML does not need to be quoted most of the time, neither for keys nor values, which greatly improves readability. This is a big difference from JSON, where all strings and identifiers (keys) must be quoted.

4.1.5 Universality

JSON is widely spread, being a common standard for many applications on the web. This has lead to most browsers, and many web frameworks, adding built in support for the format. YAML on the other hand is currently not a common format for data exchange on the web. The relatively high complexity of YAML results in higher requirements for implementations.

4.1.6 Syntax

Table 4.2 showcases the differences found from the syntax comparison between YAML and JSON, using earlier research which can be found in the background section. The 1.2 version of the YAML specification brought increased compatibility with JSON. They are not completely compatible however, as YAML requires whitespace between mappings and key-value pairs, which can be seen in table 4.3. This example shows that JSON can be written to be valid YAML, but it does not work the other way round[21]. Correctly formatted JSON (with whitespace) can be read by a YAML parser adhering to the v1.2 standard due to the fact that YAML also incorporates the alternative inline syntax.

4.1.7 Scope of use

JSON is designed to be a light-weight data exchange format, and is especially common in web based applications. YAML, on the contrary, with its design goal of

Table 4.2: Syntax comparison

Type	JSON	YAML
Comments	Not allowed in the current specification, previously possible.	Denoted with a hash/number sign, continues for the rest of the line.
Hierarchy	Objects and arrays can be nested, and are denoted by braces and brackets, respectively.	Mappings and sequences can be nested. Hierarchy is determined by indentation level.

Arrays	`["first", "second", 3]`	`- first`
		`- second`
		`- 3`
		Alt. `[first, second, 3]`

Objects	`{"object": {`
	`  "a": one,`
	`  "b": 2`
	`}}`
		`mapping:`
		`  a: one`
		`  b: 2`
		Alt. `{a: one, b: 2}`

Documents	Root node must be an array or object. Does not support multiple documents within a transmission.	Root note can be any valid data type. New documents in a transmission is denoted by three dashes. Repeated nodes are defined with ampersand, then referenced to with an asterisk.
Strings	Must be double quoted. Allows character (tabs, newlines, etc.) escaping with backslash as the escape character.	Does not require quoting but supports both single and double quotes (same functionality as JSON). Also provides two different block notations.
Numbers	Floating point numbers in scientific notation. Infinity is not permitted.	Built-in support for integers, floating-point, octal and hexadecimal numbers.

Table 4.3: JSON and YAML compatibility

```
# Valid JSON and  YAML.
{"name": "Malin", "school": "KTH", year: 2008}

# Valid JSON but invalid YAML due to missing whitespace
{"name":"Malin", "school": "KTH",year: 2008}

# Valid YAML, invalid JSON due to lack of quotes around strings
{name: Malin, school: "KTH", year: 2008}
```

high readability along with support for additional complex features, is foremost used for files meant to be manipulated by humans, such as configuration files. It offers extended possibilities of describing complex structures, which can be considered redundant in just plain data exchange.

One could consider YAML to be a good substitute for JSON in many tasks, but the required whitespace and indentation doesn't do much good when the data is only meant to be parsed by another computer. It only results in longer processing times, for no apparent reason. In terms of serialization, there is often no reason to implement more advanced functionalities. The only required task would be to transmit some data set, a mission which JSON performs perfectly fine in most cases, and many times more efficient (in terms of processing times) than YAML. This is mostly due to the fact that YAML is more complex in its structure, which affects parsing speed negatively, compared to JSON where parsing is done quite fast and efficient. This seems to be an important factor to programmers today, making JSON more commonly used and widespread.

4.2 Performance

This section aims to present the test results from the performance tests described in the Methods chapter. Testing was conducted on two different data sets, as described in 3.2. The results are grouped by attribute tested, and results are presented in tables showcasing the performance of each method for simple and complex data sets, respectively.

As the JSON standard does not explicitly require indentation (or even spacing between identifiers and values), there are two options when generating JSON. The first one is to simply output only what's necessary, which means smaller data size (as no whitespace has to be written). This does however minimize readability, which is not the case for the second option - to have the serializer include indendation and whitespace (where approriate) to maximize readability. Both cases are included in the tests, as the JSON implementation used provides methods for both compact (`JSON.generate`) and formatted (`JSON.pretty_generate`) generation.

4.2.1 Serialization performance

Table 4.4: Serialization (dump) performance

Method	Simple	Complex
`JSON.generate`	0.1550s	0.5830s
`JSON.pretty_generate`	0.1470s	0.6060s
`YAML.dump`	2.4531s	3.4732s

Execution times in seconds for each method on both data sets.

The difference in serialization process performance of both implementations is very noticeable, as shown in table 4.4. Both JSON generators are relatively equivalent, while the YAML generation is much slower for both sets of data (about 16 times slower for the simple set, and 6 times for the complex).

Something worth noting here is that the YAML implementation handles complex data relatively better, as the time taken to dump it is only about 1.5 times the simple data set run - compared to JSON which took 4 times longer on the complex data.

4.2.2 Deserialization performance

Table 4.5: Deserialization (load) performance

Method	Simple	Complex
`JSON.parse`	0.0440s	0.0790s
`YAML.load`	0.2750s	0.3360s

Execution times in seconds for each method on both data sets.

The execution times measured for the deserialization (load) process shows results similar to the serialization process, which can be seen in table 4.5. Both implementations are much faster at generating data structures from a serialized string than doing the opposite. YAML is also slower here, but only 6 (simple data set) and 4 (complex) times in this case.

4.2.3 Serialization output size

Table 4.6: Serialized data size

Method	Simple	Complex
`JSON.generate`	841.71kB	6266.31kB
`JSON.pretty_generate`	1124.92kB	6743.35kB
`YAML.dump`	831.95kB	6908.54kB

Size (in kilobytes) of the output generated by each method on both data sets.

As can be seen in table 4.6, YAML produces the most compact output for the simple data set - even smaller than the compact JSON output. The prettily formatted JSON is considerably larger than the YAML (and the compact JSON, obviously) output. The smaller size of the YAML output is due to the fact that quotes in the JSON output stands for a noticeable percent (roughly 11%) of the output.

A change can be seen for the complex data set, where the YAML output is larger than both variants of JSON. As the document hierarchy gets more complex, with deeper nesting being added,the amount of whitespace needed for YAML to correctly indent everything grows noticeably.

Chapter 5
Conclusions

5.1 Theoretical research

5.1.1 Design goals

The primary design goals for JSON, to be a simple and effective data exchange format, but also being easy to generate and parse seems to accord with the research conducted in this study. It is widely used on the net, and is used natively available in the most common modern web browsers. The fact that it is easy to implement, and has been just that in numerous libraries in different programming languages strengthen it's usefulness. JSON meets its design goals as a simple exchange format very well, but can require extra work to function well on data sets which contain anything other than the built in types it supports.

On the contrary, there is YAML, whose design goals focuses on human readability and extendability. As has been concluded, it is clearly very easy to read thanks to the required usage of whitespace and the ability to skip surrounding quotes for strings. YAML also has the advantage of allowing comments in the document. Users can easily read and manipulate the output, which is one of the reasons as to why its often used for configuration files and the like.

5.1.2 Functionality

JSON only supports a simple hierarchy, built through associative arrays and lists. Extensions exist which enables simulated object references, but this requires some work and will not be discussed in this comparison. YAML natively supports object references and relational trees. This enables it to present cyclic data structures and deep hierarchies easily. Extended data typing, for both custom types and general data such as date types, is also implemented, which facilitates its aims to produce human readable files from complicated structures. JSON lacks support for more complex data types and does not support object references at all. Most advanced data types can be expressed as a combination of the basic types available in JSON however. An example would be how dates are incorporated as strings.

Simplicity factors can be considered to be affected when working with more complicated structures where human readability will be deteriorated, though it makes no great impact on effectiveness and parsing as it seems. YAML is not as widely used likely due to the fact that most data being stored or transmitted from servers to clients over the net doesn't explicitly need the extended functionality.

5.1.3 Similarity and evolution

An interesting point is that developers from both YAML and JSON somehow aims to make the languages quite similar. YAML developers included the alternative inline syntax in version 1.2, and Douglas Crockford (the founder of JSON) removed commenting and the ability to use single quotes for strings from the former JSON specification to be inline with the YAML specification.

The YAML specification is still being revised based on continous user input from the YAML mailing list. This can affect universality and usability in future versions of the specification in an positive way, possibly making it a more common format on the net. The JSON standard is a one version standard, but it has been revised. The founder never intended it to be a evolving standard however, which is the reason for why a version number is not included [4].

5.2 Performance

The tests shows that the performance of the JSON and YAML implementations being tested greatly differ. JSON generation and parsing is faster for all sets of data tested. A simple reason for this (not necessarily exclusive to the implementations being used) could be the fact that YAML as a format is much more complex due to the additional features available, and therefore requires more processing when loading and dumping data. This can be seen by investigating the serialization process, for which the YAML specification describes four different stages - whereas JSON only requires a single pass of the data.

One general explanation behind the YAML implementation's seemingly bad performance is that it does a more thorough work in the general serialization process. This includes inspecting objects, inserting tags for custom data types and taking advantage of the alternative string block notations in YAML. The JSON implementation used only supports serialization to and from the basic data types, and will fail for custom objects unless a `to_json` method has been defined. The deserialization process does not handle custom types at all, which means that the `JSON.parse` method is limited to only returning an array or hash with the deserialized values.

Comparing the execution times between simple and complex data yields some interesting information. YAML seems to handle a growing data set with deeper hierarchies relatively better than JSON. It is possible that an even bigger data set could favour YAML even more, possibly ending up as the faster implementation for data that big.

Looking at the test results and reflecting on the theoretical research, it is clear that JSON is the favorable serialization format when speed is the most important factor. As such, JSON is the format to recommend unless the data is very complex or requires features not available to JSON, such as object references/relations or the ability to deserialize custom objects into their original form (not generic objects or arrays). In these cases YAML is a good alternative.

5.3 Format of choice

Theoretical comparison has shown both languages to be very useful, but in different ways. They seem to meet their formerly stated design goals rather well. One could consider to use them for the same task, and it would likely work well. Both formats excels in different aspects though, and the choice of format to use in a project greatly depends how the three factors below are prioritized.

5.3.1 Functionality

YAML features some functionality missing in JSON. If the data to be serialized includes object references and these are to be preserved one will have to use YAML, or JSON together with an extension providing support for this (such as `dojox.json.ref`[19]). YAML also includes the ability to tag values as specific data types, either to map data to user defined types or to explicitly cast values other basic types. This allows YAML implementations to directly transform data structures in the serialized document into native objects in the programming environment. JSON lacks this ability, and such solution will have to be developed separately if complete serialization is desired.

5.3.2 Readability

YAML is the recommended choice if high human readability is desired as even the "pretty" output of the JSON generator gets increasingly harder to read for a human as the data set grows. This factor is of least importantance when the serialized data is only meant to be transferred and parsed by another computer.

5.3.3 Performance

The performance testing proved the JSON implementation to be many times faster than YAML for both serialization (dumping) and deserialization (loading). The complexity of the YAML processing was most likely the biggest reason behind this. The relevancy of this depends on how critical processing speed is in the project, as both implemenations processed the data in respectable times.

References

[1] Victor Hallberg, 2011. *Ruby script used to benchmark JSON and YAML.* <http://github.com/mogelbrod/json-yaml-benchmark>

[2] Ecma International, 2009. *ECMAScript Language Specification.* 5th edition. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>

[3] Crockford D. *Introducing JSON.* (Accessed 2011-02-07). <http://www.json.org/index.html>

[4] Crockford D. *The JSON-saga.* (Accessed 2011-02-21). <http://developer.yahoo.com/yui/theater/video.php?v=crockford-json>

[5] JSON.org. *JSON: The Fat-Free Alternative to XML.* (Accessed 2011-03-28). <http://www.json.org/xml.html>

[6] Ben Kiki O. *YAML Ain't Markup Language (YAML) Version 1.2.* (Accessed 2011-02-21). <http://www.yaml.org/spec/1.2/spec.html>

[7] Sureau D. *JSON - JavaScript Object Notation.* (Accessed 2011-02-07). <http://www.xul.fr/ajax-javascript-json.html>

[8] Cline M. *Serialization and Unserialization.* (Accessed 2011-02-07). <http://www.parashift.com/c++-faq-lite/serialization.html#faq-36.3>

[9] Oracle. *Java Object Serialization Specification: 1 - System Architecture.* (Accessed 2011-02-22). <http://download.oracle.com/javase/6/docs/platform/serialization/spec/serial-arch.html>

[10] MSDN (Microsoft). *Object Serialization in .NET.* (Accessed 2011-02-22). <http://msdn.microsoft.com/en-us/library/ms973893.aspx>

[11] MSDN (Microsoft), 2007. *An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET.* (Accessed 2011-03-28). <http://msdn.microsoft.com/en-us/library/bb299886.aspx>

[12] Google. *Using JSON in the Google Data Protocol.* (Accessed 2011-02-21). <http://code.google.com/intl/sv-SE/apis/gdata/docs/json.html>

[13] Wikipedia. *JSON.* (Accessed 2011-02-09). <http://en.wikipedia.org/wiki/JSON>

[14] *Ruby Programming Language.* (Accessed 2011-03-24). <http://www.ruby-lang.org/>

[15] Morin M, 2010-08-26. *What is Ruby?* (Accessed 2011-03-24). <http://ruby.about.com/od/beginningruby/a/WhatIsRuby.htm>

[16] Ruby-doc. *Module: Marshal*. (Accessed 2011-02-22).
<http://www.ruby-doc.org/core/classes/Marshal.html>

[17] Frank F. *JSON implementation for Ruby*. (Accessed 2011-02-21).
<http://flori.github.com/json/>

[18] Ruby-doc (Ruby on Rails). *Module: YAML*. (Accessed 2011-04-12).
<http://corelib.rubyonrails.org/classes/YAML.html>

[19] Zyp K. *Documentation - dojox.json.ref*. (Accessed 2011-04-12).
<http://dojotoolkit.org/reference-guide/dojox/json/ref.html>

[20] Zyp K. *JSON Schema Media Type*. (Accessed 2011-04-12).
<http://tools.ietf.org/html/draft-zyp-json-schema-03>

[21] Almaer D, 2005-11-21. *JSON == YAML? It's getting closer to truth*.
(Accessed 2011-04-13).
<http://ajaxian.com/archives/json-yaml-its-getting-closer-to-truth>

[22] The Internet Society, 1998. *Uniform Resource Identifiers (URI)*.
(Accessed 2011-06-13).
<http://www.ietf.org/rfc/rfc2396.txt>