YamlTestingSuite

Search

(You are AnonymousGnome)

[TableOfContents]

Preface

The cornerstone of the YAML project is the YAML Specification. The specification was created with the intent of giving implementors a guide to writing YAML compliant processors. Unfortunately the spec has grown in size, such that it is no longer readily comprehensible to mere mortals. It is even a challenge to decipher by the very folk who created it.

The parsing productions are coded in an extended BNF which is designed for defining parsers. Ironically the EBNF productions only reside in an HTML format, rendering them useless for computer consumption.

This is not to say that the YAML spec is not important. On the contrary, it is a very complete and well thought out definition of the language. But it is not the ideal place for most programmers to start out writing a YAML processor.

What then should an implementor look to? The answer is a well defined, well organized testing suite. In many ways, a serialization language processor can be treated as a black box. You define a set of inputs and expected outputs. If the processor can produce an expected output, then it is a valid processor for that test case. This is the idea behind the YAML Testing Suite (YTS). To define tests for as many YAML situations as we can think of.

In a very real way, this test suite becomes a defacto spec for YAML. The tests, of course, must always jive with the canonical spec. But this cross-checking can be left to the few YAML immortals. The YTS also becomes a way for the language designers to test their "theory". And it gives implementors a visual, case-by-case way to digest the meat of the spec.

The YTS may never be complete, but will instead asymtotically approach completeness. The YTS should be organized in such a manner that there is an order to the tests. The most basic YAML functionality should be defined first, and followed progressively in importance to the more esoteric parts of the language. This will give programmers who are new to YAML a clear path of implementation.

A YTS project was started in August of 2002. It had tests for Ruby, Python and Perl implementations. The test suite defined here is a refactoring and extension of the original. The original YTS only concentrated on YAML documents that load correctly. But there are several other facets to testing YAML, including dumping tests, configuration, and failures.

The most novel change in the new YTS is that the entire process will be open to the public. The canonical versions of the test files will reside right here on this wiki. The tests will be categorized by various properties and put into separate

pages. The various pages will be indexed in a specific order on one wiki page: YamlTestingSuiteIndex. A simple script to download the wiki pages into test files is available at YamlTestingSuiteFetcher.

By opening this up to the public, we allow the tests to be constantly refactored towards a more understandable implementation guide. Since the tests themselves are all formatted in YAML (see below) it will be easy for people to generate the content into a reference website, or generate reports and charts on cross-language compliance.

To be safe, the files will be backed up by a cron job into a version control system. But the wiki will be considered the true source. I invite the entire YAML community to contribute to this effort. At this point in time I believe it is the most valuable use of resources for moving YAML forward.

YTS Implementation

YAML is by design the ultimate human visualization for common textual data. Since tests can be thought of merely as chunks of input and output data, it makes sense to define the tests themselves in YAML. This creates an obvious bootstrapping dilemma. How do we test a YAML parser with tests written in YAML. The solution is to use a very restrictive subset of YAML that is trivial to parse and load. This section will discuss the limitations of that subset and suggest a schema for the various types of YAML tests.

YAML Subset Restrictions

The YTS streams will be compliant with the YAML 1.0 specification, but will be constrained to the following syntax:

- **Multiple Document Stream with —\n Separators**:

Most test files will contain several related test cases. Each document represents one test case. Since anchors and types won't be supported, the separator line will always be just '—'.

- **Top Level Mapping**:

Each document in the stream will be a simple mapping with a known set of keys. There will be a few specific schemas for these mappings, according to the types of tests that are defined.

- **Simple Keys**:

Top level mapping keys will be a known set of short words or phrases. '-' will be used instead of a space if a key has more then one word.

- **One Level Deep**:

The mapping defining a test will only have scalars for values. No mapping entry will have a collection as a value. That means the the Loader doesn't need to support nested collections.

- **Plain Scalars**:

Scalars will only be of the plain and literal forms. Plain scalars are unquoted and don't span more than one line. Any scalar that needs more than one line should be a literal block.

- **Literal Blocks**:

The literal block must be implemented in full. This includes '+', '-' and explicit indentation indicators. Fortunately this is rather trivial to do.

- **Full Line Comments**

The YAML Loader should support full line comments because they are useful for commenting out tests. This requirement could be optional. Comments can also be used by wiki passersby to make useful commentary or suggestions on a certain test, without worrying about messing up the test.

That's all. Tests should not use any other features of YAML than those defined above. No sequences and no flow collections. No directives, anchors, aliases or type URIs.

Types of Tests

There are a several different angles that should be taken to properly test a YAML implementation. This section gives a brief overview of the purpose and operation of each type of test. The specifics of how to create a certain type of test is discussed below under "Test Schemas".

- **Load-Expect** - This type of test use a piece of YAML as its input, and generally a piece of the native languages source code as its expected result. The test is performed by loading the YAML and evaluating (running) the source code and comparing the result.

*

- NB - eval'ing information from a publicly accessible wiki is
- potentially very dangerous, do so at your own risk.

*

The comparison can be performed by an in memory comparsion algorithm or by dumping both results to a trusted/robust/canonical serializing format. (For example, Perl uses :Dumper to compare objects). This test can be classified as YN (YAML to Native). It is also possible to extend this test to be YNYN, by redumping/reloading the loaded object to see if it produces the same object. This is known as *roundtripping* in YAML jargon.

- **Dump-Expect** - This is the opposite of Load/Expect. The input is a piece of source code and the expected value is a piece of YAML. Often this requires that certain API options be set, like indentation width for example. This type of test can be classified as NY. It can extended to

NYN (roundtripping).

- **Error-Expect** - This test takes as it input *either* a piece of YAML or a piece of source code. If it is a piece of YAML, a load is performed, otherwise the source code is evaluated and the result is dumped. The operation is supposed to cause an error. There is an **expect** clause containing a string pattern which should be present in the resulting error message.

- **Parse-Pass** - A simpler parsing test. A piece of YAML is fed into the parser. It must not cause an error. This type of test is weaker than Load/Expect, but may be necessary for tests used by libyaml.

- **Parse-Fail** - This is the opposite of Parse/Pass. The specified YAML must cause a parse error. This is useful to test for edge cases where text looks like valid YAML, but is not.

Wiki Test File Organization

A single test file will reside on a single wiki page. The page should be named with a Yts prefix. Like YtsSpecExamples. A test file should give good coverage of a specific YAML concept. Like 'single quoted strings' for example. A single test file can use any of the above test types to fully describe the topic at hand. The tests will be listed in order from most basic to most esoteric in the YamlTestingSuiteIndex page. There can be alternate index pages to indicate what tests a given implementation passes, like YamlTestingSuitePerlIndex.

Test Schemas

A test schema is simply a definition of what entries (keys) must/can be present in a given test type. It also describes what each of these entries mean and how they should be used.

This section defines the entries that are common to all tests:

- **name** - Required - A short (1-5 words) name for the test. Should be a plain scalar value.

- **brief** - Recommended - A brief description of the problem. One or two paragraphs. Should use a literal block. Since this is documentation, a folded form isn't necessary (this was the former style). An application can fold the literal on its own if need be.

- **author** - Optional - The author is the person who first submitted the test. Should be a plain scalar of the form 'Foo Barson <foo@barson.com>' or a wiki-link style 'FooBarson'.

- **notes** - Optional - A place for persistent implementation notes. These notes might get printed in a reference manual for instance. Transient notes about the test should go in comments and may deleted when no longer relevant.

Load-Expect

- **type** - Required - Must be set to 'load-expect'.
- **yaml** - Required - The stream of YAML to be parsed and loaded. Must be a literal block.
- **$IMPL** - Required -This indicates the source code for a particular implementation. The $IMPL key is actually 'perl', 'python', 'ruby' or whatever other implementation the test supports. There will generally be multiple $IMPL entries.
- **$IMPL-config** - Optional - Sometimes a test needs additional code to work properly. One of these keys would look like 'ruby-conf', and would point to some Ruby code that should be evaluated before the load operation.
- **no-round-trip** - Optional - By default, a load-expect test must YNYN roundtrip to pass. Sometimes this needs to be defeated for a particular implementation at a particular stage. The value of this entry should be a plain scalar that is a list of implementations separated by spaces.

Example:

```
---
name: Simple Block Sequence
brief: |
    This test loads a simple sequence of scalars using
    the bulleted block form.
type: load-expect
yaml: |
    ---
    - one
    - two
    - 42
perl5: |
    [ one => two => 42 ]
python: |
    [ 'one', 'two', 42 ]
ruby: |
    [ 'one',
      'two',
      42
    ]
no-round-trip: perl python
author: BrianIngerson
```

Dump-Expect

- **type** - Required - Must be set to 'dump-expect'.
- **$IMPL** - Required -This indicates some source code for a particular implementation. The value returned by this code is dumped.
- **yaml** - Required - The stream of YAML that is the expected result of the

dump.

- **$IMPL-config** - Optional - Additional language specific code or options to control the dump.
- **no-round-trip** - Optional - By default, a dump-expect test must NYN roundtrip to pass. Sometimes this needs to be defeated for a particular implementation at a particular stage. The value of this entry should be a plain scalar that is a list of implementations separated by spaces.

Error-Expect

- **type** - Required - Must be set to 'error-expect'.
- **yaml** - Optional - A stream of YAML that will be loaded and is expected to produce an error. If not used, then the **$IMPL** entry must be used.
- **$IMPL** - Optional - This indicates some source code for a particular implementation. The value returned by this code is dumped and expected to produce an error. If not used, then the **yaml** entry must be specified.
- **$IMPL-error** - A pattern that must match the error message produced.
- **$IMPL-config** - Optional - Additional implementation specific code or options to control the dump.

Parse-Pass

- **type** - Required - Must be set to 'parse-pass'.
- **yaml** - Required - The stream of YAML to be parsed. Must be a literal block.

Parse-Fail

- **type** - Required - Must be set to 'parse-fail'.
- **yaml** - Required - The stream of YAML to be parsed. Must be a literal block.

Current Implementations

Each implementation has an identifier that is used to configure its tests. The identifier is generally the name of the language of the impementation. The following identifiers are in use for implementations currently in production:

- perl - YamlPm
- ruby - YamlForRuby
- python - PurePythonParserForYaml
- libyaml - LibYaml

Implementation authors are responsible for designing a test harness to run the

various tests.