bowtie

Release 2024.5.17

Julian Berman

May 15, 2024

CONTENTS

Bowtie is a *meta*-validator of the JSON Schema specification, by which we mean it coordinates executing *other* validator implementations, collecting and reporting on their results.

It has a few parts:

- a command line tool which can be used to access any of the 26 JSON Schema implementations integrated with it. Use it to validate any schema and instance against any implementation.
- a report, accessible at https://bowtie.report/, showing how well each implementation complies with the specification and which tests they fail (if any)
- a protocol which new implementations can integrate with in order to have their implementation be supported by Bowtie

It's called Bowtie because it fans in lots of JSON then fans out lots of results: `>·<`. Looks like a bowtie, no? Also because it's elegant – we hope.

If you're just interested in how implementations stack up against each other, you can find the most up to date report at https://bowtie.report/.

CHAPTER

ONE

CLI

Bowtie is a versatile tool which you can use to investigate any or all of the implementations it supports. Below are a few sample command lines you might be interested in.

Running Commands Against All Implementations

Many of Bowtie's subcommands take a `-i / --implementation` option which specifies which implementations you wish to run against. In general, these same subcommands allow repeating this argument multiple times to run across multiple implementations. In many or even most cases, you may be interested in running against *all* implementations Bowtie supports. For somewhat boring reasons (partially having to do with the GitHub API) this "run against all implementations" functionality is slightly nontrivial to implement in a seamless way, though doing so is nevertheless tracked in this issue.

In the interim, it's often convenient to use a local checkout of Bowtie in order to list this information.

Specifically, all supported implementations live in the `implementations/` directory, and therefore you can construct a string of `-i` arguments using a small bit of shell vomit. If you have cloned Bowtie to `/path/to/bowtie` you should be able to use `$(bowtie filter-implementations | sed 's/^/-i /')` in any command to expand out to all implementations. See *below* for a full example.

1.1 Examples

1.1.1 Validating a Specific Instance Against One or More Implementations

The *bowtie validate* subcommand can be used to test arbitrary schemas and instances against any implementation Bowtie supports.

Given some collection of implementations to check – here perhaps two Javascript implementations – it takes a single schema and one or more instances to check against it:

```
$ bowtie validate -i js-ajv -i js-hyperjump <(printf '{"type": "integer"}') <(printf 37)
↪<(printf '"foo"')
```

Note that the schema and instance arguments are expected to be files, and that therefore the above makes use of normal shell process substitution to pass some examples on the command line.

Piping this output to *bowtie summary* is often the intended outcome (though not always, as you also may upload the output it gives to https://bowtie.report/ as a local report). For summarizing the results in the terminal however, the above command when summarized produces:

```
$ bowtie validate -i js-ajv -i js-hyperjump <(printf '{"type": "integer"}') <(printf 37)
↪<(printf '"foo"') | bowtie summary
2023-11-02 15:43.10 [debug    ] Will speak                     dialect=https://json-
↪schema.org/draft/2020-12/schema
2023-11-02 15:43.10 [info     ] Finished                       count=1
                                        Bowtie

 Schema

│                     │                                                              │
│ {                   │   Instance   ajv (javascript)   hyperjump-jsv (javascript)   │
│   "type": "integer" │  ──────────────────────────────────────────────────────────  │
│ }                   │   37         valid              valid                        │
│                     │   "foo"      invalid            invalid                      │
│                     │                                                              │
└──────────────────────────────────────────────────────────────────────────────────
                                    2 tests ran
```

1.1.2 Running a Single Test Suite File

To run the draft 7 `type`-keyword tests on the Lua `jsonschema` implementation, run:

```
$ bowtie suite -i lua-jsonschema https://github.com/json-schema-org/JSON-Schema-Test-
↪Suite/blob/main/tests/draft7/type.json | bowtie summary --show failures
```

1.1.3 Running the Official Suite Across All Implementations

The following will run all Draft 7 tests from the official test suite (which it will automatically retrieve) across all implementations supporting Draft 7, showing a summary of any test failures.

```
$ bowtie suite $(bowtie filter-implementations | sed 's/^/-i /') https://github.com/json-
↪schema-org/JSON-Schema-Test-Suite/tree/main/tests/draft7 | bowtie summary --show␣
↪failures
```

1.1.4 Running Test Suite Tests From Local Checkouts

Providing a local path to the test suite can be used as well, which is useful if you have local changes:

```
$ bowtie suite $(bowtie filter-implementations | sed 's/^/-i /') ~/path/to/json-schema-
↪org/suite/tests/draft2020-12/ | bowtie summary --show failures
```

1.1.5 Checking An Implementation Functions On Basic Input

If you wish to verify that a particular implementation works on your machine (e.g. if you suspect a problem with the container image, or otherwise aren't seeing results), you can run *bowtie smoke*. E.g., to verify the Golang `jsonschema` implementation is functioning, you can run:

```
$ bowtie smoke -i go-jsonschema
```

1.2 Enabling Shell Tab Completion

The Bowtie CLI supports tab completion using the `click module's built-in support`. Below are short instructions for your shell using the default configuration paths.

Bash

Zsh

Fish

Add this to `~/.bashrc`:

```
$ eval "$(_BOWTIE_COMPLETE=bash_source bowtie)"
```

Add this to `~/.zshrc`:

```
$ eval "$(_BOWTIE_COMPLETE=zsh_source bowtie)"
```

Add this to `~/.config/fish/completions/bowtie.fish`:

```
$ _BOWTIE_COMPLETE=fish_source bowtie | source
```

This is the same file used for the activation script method below. For Fish it's probably always easier to use that method.

Using `eval` means that the command is invoked and evaluated every time a shell is started, which can delay shell responsiveness. To speed it up, write the generated script to a file, then source that.

Bash

Zsh

Fish

Save the script somewhere.

```
$ _BOWTIE_COMPLETE=bash_source bowtie > ~/.bowtie-complete.bash
```

Source the file in `~/.bashrc`.

```
$ . ~/.bowtie-complete.bash
```

Save the script somewhere.

```
$ _BOWTIE_COMPLETE=zsh_source bowtie > ~/.bowtie-complete.zsh
```

Source the file in `~/.zshrc`.

```
$ . ~/.bowtie-complete.zsh
```

Save the script to `~/.config/fish/completions/bowtie.fish`:

```
$ _BOWTIE_COMPLETE=fish_source bowtie > ~/.config/fish/completions/bowtie.fish
```

After modifying your shell configuration, you may need to start a new shell in order for the changes to be loaded.

1.3 Reference

1.3.1 bowtie

A meta-validator for the JSON Schema specifications.

Bowtie gives you access to the JSON Schema ecosystem across every programming language and implementation.

It lets you compare implementations either to each other or to known correct results from the official JSON Schema test suite.

```
bowtie [OPTIONS] COMMAND [ARGS]...
```

Options

`--version`

Show the version and exit.

`-L, --log-level` `<log_level>`

How verbose should Bowtie be?

Default
`warning`

Options
debug | info | warning | error | critical

If you don't know where to begin, `bowtie validate --help` or `bowtie suite --help` are likely good places to start.

Full documentation can also be found at https://docs.bowtie.report

badges

Generate Bowtie badges for implementations using a previous Bowtie run.

Will generate badges for any existing dialects, and ignore any for which a report was not generated.

```
bowtie badges [OPTIONS]
```

Options

--site <site>

The path to a previously generated collection of reports, used to generate the badges.

Default
`site`

filter-dialects

Output dialect URIs matching a given criteria.

If any implementations are provided, filter dialects supported by all the given implementations.

```
bowtie filter-dialects [OPTIONS]
```

Options

-i, --implementation <IMPLEMENTATION>

A container image which implements the bowtie IO protocol.

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-d, --dialect <URI_OR_NAME>

Filter from the given list of dialects only.

-l, --latest

Show only the latest dialect.

-b, --boolean-schemas, -B, --no-boolean-schemas

If provided, show only dialects which do (or do not) support boolean schemas. Otherwise show either kind.

filter-implementations

Output implementations which match the given criteria.

Useful for piping or otherwise using the resulting output for further Bowtie commands.

```
bowtie filter-implementations [OPTIONS]
```

Options

-i, --implementation <IMPLEMENTATION>

A container image which implements the bowtie IO protocol.

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-d, --supports-dialect <URI_OR_NAME>

Only include implementations supporting the given dialect or dialect short name.

-l, --language <LANGUAGE>

Only include implementations in the given programming language

Options
c++ | clojure | cpp | dotnet | go | java | javascript | js | kotlin | lua | php | python | ruby | rust | scala | ts | typescript

info

Show information about a supported implementation.

```
bowtie info [OPTIONS]
```

Options

-i, --implementation <IMPLEMENTATION>

A container image which implements the bowtie IO protocol.

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-f, --format <format>

What format to use for the output

Default
`pretty if stdout is a tty, otherwise JSON`

Options
json | pretty | markdown

--schema

Show the JSON Schema for this command's JSON output.

run

Run test cases written directly in Bowtie's testing format.

This is generally useful if you wish to hand-author which schemas to include in the schema registry, or otherwise exactly control the contents of a test case.

```
bowtie run [OPTIONS] [INPUT]
```

Options

-i, --implementation <IMPLEMENTATION>

Required A container image which implements the bowtie IO protocol.

-D, --dialect <URI_OR_NAME>

A URI or shortname identifying the dialect of each test. Possible shortnames include: 2019, 2019-09, 201909, 2020, 2020-12, 202012, 3, 4, 6, 7, draft2019-09, draft201909, draft2020-12, draft202012, draft3, draft4, draft6, draft7.

-k, --filter <GLOB>

Only run cases whose description match the given glob pattern.

-x, --fail-fast

Stop running immediately after the first failure or error.

--max-fail <COUNT>

Stop running once COUNT tests fail in total across implementations.

--max-error <COUNT>

Stop running once COUNT tests error in total across implementations.

-S, --set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
`False`

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie's JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

Arguments

INPUT

Optional argument

smoke

Smoke test implementations for basic correctness against Bowtie's protocol.

```
bowtie smoke [OPTIONS]
```

Options

-i, --implementation <IMPLEMENTATION>

A container image which implements the bowtie IO protocol.

-T, --read-timeout <SECONDS>

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-q, --quiet

Don't print any output, just exit with nonzero status on failure.

-f, --format <format>

What format to use for the output

Default
`pretty if stdout is a tty, otherwise JSON`

Options
json | pretty | markdown

--schema

Show the JSON Schema for this command's JSON output.

statistics

Show summary statistics for a previous report.

```
bowtie statistics [OPTIONS] [REPORT]
```

Options

-f, --format <format>

What format to use for the output

Default

`pretty if stdout is a tty, otherwise JSON`

Options

json | pretty | markdown

--schema

Show the JSON Schema for this command's JSON output.

--quantiles <n>

How many quantiles should be emitted for the compliance numbers? Computing quantiles only is sensical if this number is more than the number of implementations reported on. By default, we compute quartiles.

Arguments

REPORT

Optional argument

suite

Run the official JSON Schema test suite against any implementation.

Supports a number of possible inputs:

- file paths found on the local file system containing tests, e.g.:
  - `{PATH}/tests/draft7` to run the draft 7 version's tests out of a local checkout of the test suite
  - `{PATH}/tests/draft7/foo.json` to run just one file from a checkout
- URLs to the test suite repository hosted on GitHub, e.g.:
  - `https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/main/tests/draft7/` to run a version directly from any branch which exists in GitHub
  - `https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/main/tests/draft7/foo.json` to run a single file directly from a branch which exists in GitHub
- short name versions of the previous URLs (similar to those providable to *`bowtie validate --dialect`*, e.g.:
  - 7, to run the draft 7 tests directly from GitHub (as in the URL example above)

```
bowtie suite [OPTIONS] INPUT
```

Options

-i, --implementation `<IMPLEMENTATION>`

Required A container image which implements the bowtie IO protocol.

-k, --filter `<GLOB>`

Only run cases whose description match the given glob pattern.

-x, --fail-fast

Stop running immediately after the first failure or error.

--max-fail `<COUNT>`

Stop running once COUNT tests fail in total across implementations.

--max-error `<COUNT>`

Stop running once COUNT tests error in total across implementations.

-S, --set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
`False`

-T, --read-timeout `<SECONDS>`

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie's JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

Arguments

INPUT

Required argument

summary

Generate an (in-terminal) summary of a Bowtie run.

```
bowtie summary [OPTIONS] [REPORT]
```

Options

-f, --format <format>

What format to use for the output

Default
`pretty if stdout is a tty, otherwise JSON`

Options
json | pretty | markdown

--schema

Show the JSON Schema for this command's JSON output.

-s, --show <show>

Configure whether to display validation results (whether instances are valid or not) or test failure results (whether the validation results match expected validation results)

Default
`validation`

Options
failures | validation

Arguments

REPORT

Optional argument

validate

Validate instances under a schema across any supported implementation.

```
bowtie validate [OPTIONS] SCHEMA [INSTANCES]...
```

Options

-i, --implementation <IMPLEMENTATION>

Required A container image which implements the bowtie IO protocol.

-D, --dialect <URI_OR_NAME>

A URI or shortname identifying the dialect of each test. Possible shortnames include: 2019, 2019-09, 201909, 2020, 2020-12, 202012, 3, 4, 6, 7, draft2019-09, draft201909, draft2020-12, draft202012, draft3, draft4, draft6, draft7.

-S, --set-schema

Explicitly set $schema in all (non-boolean) case schemas sent to implementations. Note this of course means what is passed to implementations will differ from what is provided in the input.

Default
`False`

-T, --read-timeout `<SECONDS>`

An explicit timeout to wait for each implementation to respond to *each* instance being validated. Set this to 0 if you wish to wait forever, though note that this means you may end up waiting … forever!

Default
`2.0`

-V, --validate-implementations

When speaking to implementations (provided via -i), validate the requests and responses sent to them under Bowtie's JSON Schema specification. Generally, this option protects against broken Bowtie implementations and can be left at its default (of off) unless you are developing a new implementation container.

-d, --description `<description>`

A (human-readable) description for this test case.

--expect `<expect>`

Expect the given input to be considered valid or invalid, or else (with 'any') to allow either result.

Default
`any`

Options
valid | invalid | any

Arguments

SCHEMA

Required argument

INSTANCES

Optional argument(s)

CHAPTER

TWO

USING BOWTIE IN GITHUB ACTIONS

Bowtie can be used from within GitHub Actions by using it in a GitHub workflow step. For example:

```
name: Run Bowtie
on: [push]

jobs:
  bowtie:
    runs-on: ubuntu-latest

    steps:
      - name: Install Bowtie
        uses: bowtie-json-schema/bowtie@2024.5.17
```

See also:

Workflow Syntax for GitHub Actions

for full details on writing GitHub Actions workflows

Once you have installed it, the *Bowtie CLI* will be available in successive run steps. Most commonly, you can use it to validate an instance (some data) using a specific JSON Schema implementation by adding:

```
- name: Validate Schema
  run: bowtie validate -i lua-jsonschema schema.json instance.json
```

replacing `lua-jsonschema` and the filenames with your implementation and schema of choice. For full details on the commands available, see the *CLI documentation*.

A fully working example of the above code can also be found here.

2.1 Including Bowtie Output in a Workflow Summary

Some of Bowtie's commands, notably *bowtie summary*, support outputting in markdown format using the `--format markdown` option. This can be useful for including their output in GitHub Actions' workflow summaries, e.g. to show validation results within the GitHub UI.

For example:

```
- name: Validate 37 is an Integer
  run: |
    bowtie validate -i python-jsonschema <(printf '{"type": "integer"}') <(printf '37')␣
↪| bowtie summary --format markdown >> $GITHUB_STEP_SUMMARY
```

```yaml
- name: Smoke Test a Bowtie Implementation
run: bowtie smoke -i go-jsonschema --format markdown >> $GITHUB_STEP_SUMMARY
```

See also:

Displaying a Workflow Job's Summary

for further details on `GITHUB_STEP_SUMMARY`

CHAPTER

THREE

CONTRIBUTOR'S GUIDE

Thanks for considering contributing to Bowtie, it'd be very much appreciated!

3.1 Installation

If you're going to work on Bowtie itself, you likely will want to install it using Python's `editable install functionality`, as well as to install Bowtie's testing dependencies e.g. by running:

```
$ pip install -r test-requirements.txt -e .
```

within a checkout of the Bowtie repository. This will allow you to make changes to files within Bowtie and see results without reinstalling it repeatedly.

3.2 Running the Tests

Bowtie has a small set of integration tests which ensure it works correctly on a number of cases.

You can find them in the `tests/` directory.

You can run them using `nox`, which you can install using the `linked instructions`. Once you have done so, you can run:

```
$ nox -s tests-3.11
```

to run the tests using Python 3.11 (or any other version you'd like).

There are additional environments which you can have a look through by running `nox -l`. Before submitting a PR you may want to run the full suite of tests by running `nox` with no arguments to run all of them. Continuous integration will run them for you regardless, if you don't care to wait.

3.3 Running the UI

Bowtie has a frontend interface used to view or inspect results of Bowtie test runs.

A hosted version of this UI is what powers https://bowtie.report/. If you are making changes to the UI, you can run it locally by:

- ensuring you have the pnpm package manager installed
- running `pnpm --dir frontend run start` from your Bowtie repository checkout, *or* alternatively `nox -s ui` to run the same command via `nox`

3.4 For Implementers

If you have a new (or not so new) implementation which Bowtie doesn't yet support, contributing a test harness to add it is extremely useful (to Bowtie but also hopefully to you!).

See the *harness tutorial* for details on writing a harness for your implementation, and please do send a PR to add it!

Please also feel free to ask for commit rights on Bowtie's repository, if only to work on your own implementation's harness – there's no intention to gate-keep the test harness for your implementation, so while there *is* an intentional level of homogeneity that should be kept with other harnesses, there's nonetheless a desire for development to be more open than closed.

3.5 Proposing Changes

3.5.1 What It Means to Contribute

Before mentioning a few tips about contributions, it's important to make very clear what it means to submit a contribution to Bowtie.

Bowtie is open source software, licensed under the MIT license. By submitting code or making a contribution of any kind you are *implicitly agreeing to license your contributed work under this license forever, just as Bowtie itself already is*. If you do not agree to do so you must say so explicitly, though doing so likely will mean your contribution won't be accepted. Furthermore by doing so you are also implicitly asserting you have the *right to do this* – in particular, if you have an employer and have done any portion of this work as part of your employment, you are asserting that you are able to grant these licensing terms yourself, and that your employer does not in some way own the rights themselves, and/or you assert that you have explicitly gotten permission from your employer.

None of the above should be meaningfully different from your experience with any other open source project, but occasionally new contributors may not fully understand the implications of making contributions.

(The above is of course not legal advice, nor is it a comprehensive list nor guide on what rights you are expected to have and grant when submitting pull requests.)

You may not use any large language model of any kind as part of assembling your contribution. This includes GitHub Copilot, ChatGPT or any other vendor's product(s). You may not have these models produce part of your contribution and then modify it. Any attempt to submit code generated with the help of these models will result in a ban from Bowtie's repository and the removal of your contributions.

If you are curious about the reasoning for the above, it has to do with the unclear copyright status of these models' training sets, and thereby the unclear status of code they produce, which may be regurgitated from copyrighted works. Whether these technologies are revolutionary or not, for contributions' sake, they are not allowed.

3.5.2 Found a Bug?

If you suspect you may have found a bug, feel free to file an issue after checking whether one already exists. Of course a pull request to fix it is also very welcome. Ideally, all pull requests (particularly ones that aren't frontend-focused) should come with tests to ensure the changes work and continue to work.

Continuous integration via GitHub actions should run to indicate whether your change passes both the test suite as well as linters. Please ensure it passes, or indicate in a comment if you believe it fails spuriously.

Please discuss any larger changes ahead of time for the sake of your own time! Improvements are very welcome, but large pull requests can be difficult to review or may be worth discussing design decisions before investing a lot of effort. You're welcome to suggest a change in an issue and thereby get some initial feedback before embarking on an effort that may not get merged.

3.5.3 Improving the Documentation?

Writing good documentation is challenging both to prioritize and to do well.

Any help you may have would be great, especially if you're a beginner who's struggled to understand Bowtie.

Documentation is written in `Sphinx-flavored reStructuredText`, so you'll want to at least have a quick look at the `tutorial` if you have never written documentation using it.

Feel free to file issues or pull requests as well if you wish something was better documented, as this too can help prioritize.

Building the Documentation Locally

The code for the documentation is in the docs directory of the Bowtie repository.

You can build the documentation locally by running the following command from the root of the repository

```
$ nox -s "docs(dirhtml)" -- dirhtml
```

which will generate a directory called `dirhtml` containing the built HTML.

3.5.4 Improving the Contributor's Guide Itself

Please feel free to share difficulties you had with developing on Bowtie itself. This contributor's guide is *not* meant to be a complete beginner's guide – e.g. it will not teach you how to do things which are common across all Python projects (let alone teach you Python itself). While there's no intention for it to become such a beginner's guide, it *is* completely reasonable for it to contain links to *other guides* which can help. So if you had trouble doing something in particular, please feel free to send a pull request to modify this guide, especially if you still cannot figure out how to do it.

3.5.5 Triage & PR Review

There are a number of oft-overlooked ways you can help Bowtie's development.

One in particular is helping to either triage issues (e.g. attempting to reproduce an issue someone has reported, and commenting saying you can reproduce, or even better, that you see where the issue lies).

Another is helping to review pull requests! Being a good reviewer is an important skill and also a way you can save someone else time reviewing!

3.6 Have a Question?

Please do not use the issue tracker for questions, it's reserved for things believed to be bugs, or new functionality, but please *do* feel free to open GitHub discussions on Bowtie's discussion tab. Any help you can offer to answer others' questions is of course appreciated as well. You are also welcome to ask questions on the JSON Schema Slack.

CHAPTER

FOUR

ADDING A NEW IMPLEMENTATION

The purpose of *Bowtie* is to support passing instances and schemas through a large number of JSON Schema implementations, collecting their output for comparison.

If you've written or used an implementation of JSON Schema which isn't already supported, let's see how you can add support for it to Bowtie.

Bowtie orchestrates running a number of containers, passing JSON Schema test cases to each one of them, and then collecting and comparing results across implementations. Any JSON Schema implementation will have some way of getting schemas and instances "into" it for processing. We'll wrap this implementation-specific API inside a small harness which accepts input from Bowtie over standard input and writes results to standard output in the format Bowtie expects, as shown below:

As a last step before we get into details, let's summarize some terminology (which you can also skip and refer back to if needed):

implementation
: a library or program which implements the JSON Schema specification

validation API
: the specific function(s), method(s), or similar constructs within the `implementation` which cause it to evaluate a schema against a specific instance

host language
: the programming language which a particular `implementation` is written in

test harness
test runner
: a small program which accepts `test cases` sent from Bowtie, passes them through a specific `implementation's` *validation API*, and writes the results of this validation process out for Bowtie to read

harness language
: the programming language which the `test harness` itself is written in. Typically this will match the `host language`, since doing so will make it easier to call out directly to the `implementation`.

test case
: a specific JSON schema and instance which Bowtie will pass to any `implementations` it is testing

IHOP
: the *i*nput → *h*arness → *o*utput *p*rotocol. A JSON protocol governing the structure and semantics of messages which Bowtie will send to `test harnesses` as well as the structure and semantics it expects from JSON responses sent back.

4.1 Prerequisites

- Bowtie itself, already installed on your machine
- A target `implementation`, which you do *not* necessarily need installed on your machine
- docker, podman or a similarly compatible tool for building OCI container images and running OCI containers

4.2 Step 0: Familiarization With the Implementation

Once you've installed the prerequisites, your first step is to ensure you're familiar with the implementation you're about to add support for, as well as with its `host language`. If you're its author, you're certainly well qualified :) – if not, you'll see a few things below which you'll need to find in its API documentation, such as what function(s) or object(s) you use to validate instances under schemas. If you're not, there shouldn't be a need to be an expert neither in the language nor implementation, as we'll be writing only a small wrapper program, but you definitely will need to know how to compile or run programs in the host language, how to read and write JSON from it, and how to package programs into container images.

JSON Schema Implementation "Architectures"

Most implementations of JSON Schema use a "runtime-compile-and-validate" architecture where at runtime a schema is turned into a callable which can be used to validate other language-level objects. In particular, they do *not* involve a separate out-of-band compilation process where a schema is turned into a compiled artifact that no longer references general JSON Schema behavior.

Bowtie certainly aims to support all implementations and architectures, so if you're writing a harness for a drastically different architecture, you'll need to adjust what is below to suit.

For the purposes of this tutorial, we'll write support for a Lua implementation of JSON Schema (one which calls itself simply `jsonschema` within the Lua ecosystem, as many implementations tend to). Bowtie of course already supports this implementation officially, so if you want to see the final result either now or at the end of this tutorial, it's here. If you're not already familiar with Lua as a programming language, the below won't serve as a full tutorial of course, but you still should be able to follow along; it's a fairly simple one.

Let's get a Hello World container running which we'll turn into our test harness.

Create a directory somewhere, and within it create a `Dockerfile` with these contents:

```
FROM alpine:3.19
RUN apk add --no-cache luajit luajit-dev pcre-dev gcc libc-dev curl make cmake && \
    wget 'https://luarocks.org/releases/luarocks-3.9.2.tar.gz' && \
    tar -xf luarocks-3.9.2.tar.gz && rm luarocks-3.9.2.tar.gz && \
    cd luarocks-3.9.2 && ./configure && make && make install && \
    cd .. && rm -r luarocks-3.9.2 && \
    sed -i '/WGET/d' /usr/local/share/lua/5.1/luarocks/fs/tools.lua && \
    luarocks install jsonschema && \
    apk del luajit-dev gcc libc-dev curl make cmake
WORKDIR /usr/src/myapp
COPY json.lua bowtie_jsonschema.lua .
CMD ["luajit", "bowtie_jsonschema.lua"]
```

Most of the above is slightly *more* complicated than you're likely to need for your own language, and has to do with some Lua-specific issues that are uninteresting to discuss in detail (which essentially relate to installing Lua's package manager and a library for JSON serialization).

The notable bit is we'll install our implementation and create a `bowtie_jsonschema.lua` file which is our test harness. Our container image will invoke the harness, and Bowtie will later speak to the running container.

Let's check everything works. Create a file named `bowtie_jsonschema.lua` with these contents:

```
print('Hello world')
```

and build the image (below using `podman` but if you're using `docker`, just substitute it for `podman` in all commands below):

```
podman build --quiet -f Dockerfile -t bowtie-lua-jsonschema .
```

Note: If you are indeed using `podman`, you must ensure you have set the `DOCKER_HOST` environment variable in the environment in which you invoke `bowtie`.

This ensures `bowtie` can speak the Docker API to your `podman` installation (which is needed because the API client used within Bowtie is agnostic, but speaks the Docker API, which `podman` supports as well).

Further information may be found here.

If everything went well, running:

```
podman run --rm bowtie-lua-jsonschema
```

should get you some output:

```
Hello world
```

We're off to the races.

4.3 Step 1: IHOP Here We Come

From here on out we'll continue to modify the `bowtie_jsonschema.lua` file we created above, so keep that file open and make changes to it as directed below.

Bowtie sends JSON requests (or commands) to *`test harness`* containers over standard input, and expects responses to be sent over standard output.

Each request has a `cmd` property which specifies which type of request it is. Additional properties are arguments to the request.

There are 4 commands every harness must know how to respond to, shown here as a brief excerpt from the full schema specifying the protocol:

```
"oneOf": [
  { "$ref": "tag:bowtie.report,2023:ihop:command:start" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:dialect" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:run" },
  { "$ref": "tag:bowtie.report,2023:ihop:command:stop" }
]
```

Bowtie will send a `start` request when first starting a harness, and then at the end will send a `stop` request telling the harness to shut down.

Let's start filling out a real test harness implementation by at least reacting to `start` and `stop` requests.

Note: From here on out in this document it's assumed you rebuild your container image each time you modify the harness via

```
podman build -f Dockerfile -t localhost/tutorial-lua-jsonschema .
```

Change your harness to contain:

```
local json = require 'json'

io.stdout:setvbuf 'line'

local cmds = {
  start = function(_)
    io.stderr:write("STARTING")
    return {}
  end,

  stop = function(_)
    io.stderr:write("STOPPING")
    return {}
  end,
}

for line in io.lines() do
  local request = json.decode(line)
  local response = cmds[request.cmd](request)
  io.write(json.encode(response) .. '\n')
end
```

If this is your first time reading Lua code, what we've done is create a dispatch table (a mapping from string command names to functions handling each one), and implemented stub functions which simply write to `stderr` when called, and then return empty responses.

Note: We also have configured `stdout` for the harness to be line-buffered (by calling `setvbuf`). Ensure you've done the equivalent for your host language, as Bowtie expects to be able to read responses to each message it sends even though some languages do not necessarily flush their output on each line write.

Any other command other than `start` or `stop` will blow up, but we're reading and writing JSON!

Let's see what happens if we use this nonetheless. We can pass a hand-crafted *test case* to Bowtie by running:

```
bowtie run -i localhost/tutorial-lua-jsonschema <<EOF
    {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪"instance": {}}] }
    {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description":
↪"not 37", "instance": {}}, {"description": "is 37", "instance": 37}] }
EOF
```

which if you now run should produce something like:

```
2022-10-05 15:39.59 [debug    ] Will speak dialect           dialect=https://json-
↪schema.org/draft/2020-12/schema
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
TypeError: bowtie._commands.Started() argument after ** must be a mapping, not list
```

... a nice big inscrutable error message. Our harness isn't returning valid responses, and Bowtie doesn't know how to handle what we've sent it. The structure of the protocol we're trying to implement lives in a JSON Schema though, and Bowtie can be told to validate requests and responses using the schema. You can enable this validation by passing **bowtie run** the *-V* option, which will produce nicer messages while we develop the harness:

```
bowtie run -i localhost/tutorial-lua-jsonschema -V <<EOF
{"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪"instance": {}}] }
{"description": "test case 2", "schema": {"const": 37}, "tests": [{"description": "not 37
↪", "instance": {}}, {"description": "is 37", "instance": 37}] }
EOF
2022-10-05 20:59.41 [debug    ] Will speak dialect             dialect=https://json-
↪schema.org/draft/2020-12/schema
2022-10-05 20:59.41 [error    ] Invalid response               [localhost/tutorial-lua-
↪jsonschema] errors=[<ValidationError: "[] is not of type 'object'">]␣
↪request=Start(version=1)
2022-10-05 20:59.45 [warning  ] Unsupported dialect, skipping implementation. [localhost/
↪tutorial-lua-jsonschema] dialect=https://json-schema.org/draft/2020-12/schema
{"implementations": {}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
2022-10-05 20:59.45 [info     ] Finished                       count=2
```

which is telling us we're returning JSON arrays to Bowtie instead of JSON objects. Lua the language has only one container type (table), and we've returned {} which the JSON library guesses means "empty array". Don't think too hard about Lua's peculiarities, let's just fix it by having a look at what parameters the start command sends a harness and what it expects back. The schema says:

```
{
  "description": "Sent once at program start to the implementation to indicate Bowtie is␣
↪starting to send test cases.",

  "$id": "tag:bowtie.report,2023:ihop:command:start",

  "required": ["version"],
  "properties": {
    "cmd": { "const": "start" },
    "version": {
      "description": "The version of the Bowtie protocol which is intended.",
      "$ref": "tag:bowtie.report,2023:ihop#version"
    }
  },
```

so start requests will have two parameters:

- cmd which indicates the kind of request being sent (and is present in all requests sent by Bowtie)

- version which represents the version of the Bowtie protocol being spoken. Today, that version is always 1, but that may change in the future, in which case a harness should bail out as it may not understand the requests being sent.

The harness is expected to respond with something conforming to:

```
    "response": {
      "$anchor": "response",

      "type": "object",
      "required": ["version", "implementation"],
      "properties": {
        "version": {
          "description": "Confirmation of the Bowtie version",
          "$ref": "tag:bowtie.report,2023:ihop#version"
        },
        "implementation": {
          "$ref": "tag:bowtie.report,2024:models:implementation"
        }
      }
    }
  }
}
```

which is some metadata about the implementation being tested, and includes things like:

- its name
- a URL for its bug tracker for use if issues are found
- the versions of JSON Schema it supports, identified via URIs

You can also have a look at the full schema for details on the stop command, but it essentially doesn't require a response, and simply signals the harness that it should exit.

Let's implement both requests. Change your harness to contain:

```lua
local json = require 'json'

STARTED = false

local cmds = {
  start = function(request)
    assert(request.version == 1, 'Wrong version!')
    STARTED = true
    return {
      version = 1,
      implementation = {
        language = 'lua',
        name = 'jsonschema',
        homepage = 'https://github.com/api7/jsonschema',
        issues = 'https://github.com/api7/jsonschema/issues',
        source = 'https://github.com/api7/jsonschema',

        dialects = {
          'http://json-schema.org/draft-07/schema#',
          'http://json-schema.org/draft-06/schema#',
```

(continues on next page)

(continued from previous page)

```
          'http://json-schema.org/draft-04/schema#',
        },
      },
    }
  end,

  stop = function(_)
    assert(STARTED, 'Not started!')
    os.exit(0)
  end,
}

for line in io.lines() do
  local request = json.decode(line)
  local response = cmds[request.cmd](request)
  io.write(json.encode(response) .. '\n')
end
```

We now return some detail about the implementation when responding to `start` requests (sent as JSON), including which versions of the specification it supports.

When stopping, we simply exit successfully.

If you re-run `bowtie`, you'll see now that it doesn't crash, though it outputs:

```
2022-10-11 13:44.40 [debug    ] Will speak dialect             dialect=https://json-
↪schema.org/draft/2020-12/schema
2022-10-11 13:44.40 [warning  ] Unsupported dialect, skipping implementation. [localhost/
↪tutorial-lua-jsonschema] dialect=https://json-schema.org/draft/2020-12/schema
{"implementations": {}}
```

Our harness is now properly starting and stopping, but this Lua implementation only supports versions of JSON Schema earlier than Draft 7, and Bowtie is defaulting to a newer version. Tell Bowtie we are speaking an earlier version by passing the *`--dialect`* option, i.e. `bowtie run --dialect 7 -i localhost/tutorial-lua-jsonschema`.

Tip: Any of `7`, `draft7`, or the full draft 7 meta schema URI will work to set the dialect in use.

If we yet again invoke `bowtie`, we now see something like:

```
2022-10-31 12:26.05 [debug    ] Will speak dialect             dialect=http://json-
↪schema.org/draft-07/schema#
───────────── localhost/tutorial-lua-jsonschema (stderr) ─────────────
│                                                                     │
│    luajit: bowtie_jsonschema.lua:35: attempt to call a nil value    │
│    stack traceback:                                                 │
│            bowtie_jsonschema.lua:35: in main chunk                  │
│            [C]: at 0xaaaad55c8690                                   │
│                                                                     │
───────────────────────────────────────────────────────────────────────
2022-10-31 12:26.06 [error    ] Tried to start sending test cases, but got an error.␣
↪[localhost/tutorial-lua-jsonschema]
2022-10-31 12:26.06 [error    ] No implementations started successfully!
```

which is indicating that we have yet another command to implement – the `dialect` command.

4.4 Step 2: Configuring Implicit Dialects

The JSON Schema specification generally allows schemas of the form `{"type": "object"}` – i.e. ones where the schema does not internally include a $schema keyword which would otherwise indicate the dialect of JSON Schema being used. In other words, the aforementioned schema may be treated (depending on its author's intention) as a Draft 4 schema, a Draft 7 schema, a Draft 2020-12 schema, etc. Bowtie enables specifying an intended behavior for such schemas by communicating it "out-of-band" to harnesses via the `dialect` command, which indicates to the harness: "treat schemas without `$schema` as this particular dialect (provided in the request)". The structure of this command looks like:

```
{
  "description": "Sent to indicate an 'implicit dialect' -- i.e. which dialect is␣
↪intended for schemas which do not contain $schema. May be sent multiple times by␣
↪bowtie to indicate a change in implicit dialect. Note that implementations do not have␣
↪to support processing such schemas in certain versions of the JSON Schema␣
↪specification. Harnesses should *not* attempt to pepper this behavior over, or␣
↪otherwise change the behavior of an implementation. In other words, this value should␣
↪*not* be used to mutate incoming schemas (by inserting $schema). If an implementation␣
↪does not support indicating what dialect a schema is written against unless indicated␣
↪by $schema, or if it refuses to process schemas which do not contain $schema entirely,␣
↪it should respond to this command as indicated below, and simply error when running␣
↪cases containing such schemas as it would when used normally. Regardless of the␣
↪dialect specified by this request, *explicitly* dialected schemas (which do contain
↪$schema) with a different dialect may still be sent (as long as the implementation has␣
↪signalled it supports the dialect)!",

  "$id": "tag:bowtie.report,2023:ihop:command:dialect",

  "properties": {
    "cmd": { "const": "dialect" },
    "dialect": {
      "description": "A dialect URI which has previously been recognized as supported by␣
↪the implementation's start response.",

      "type": "string",
      "format": "uri"
    }
  },
```

We need the harness to accept the incoming dialect request and configure the Lua implementation that we're wrapping to treat `$schema`-less schemas as the specified dialect – only there's a catch. This implementation actually doesn't support a mechanism for specifying how to treat these kinds of schemas.

See also:

The harness for the Rust jsonschema-rs implementation
for an example of an implementation which does support self-configuration for the `dialect` command.

In this case, the harness should flag this to Bowtie so that it is aware that the harness is unable to configure the implementation in this way, which we do by simply responding with `{"ok": false}`. Add a handler for the `dialect` command to your harness which returns that response:

```
dialect = function(_)
  assert(STARTED, 'Not started!')
  return { ok = false }
end,
```

Warning: Responding `{"ok": true}` or `false` is *not* an indication of whether an implementation supports the dialect sent.

Bowtie will never send a dialect request for a dialect that a harness does not support – information which it already knows from the `start` response we implemented *earlier*, where we specified the complete list of supported dialects for the implementation. If it ever did so this would be considered a Bowtie bug.

This `dialect` request *strictly* controls setting what an implementation harness should do with schemas that do *not* internally indicate what version they are written for; its response should signal only whether the implementation has configured itself appropriately or whether doing so is not supported.

Bowtie will *continue executing tests* even if it sees a `false` response.

Running `bowtie` now should produce:

```
2022-10-31 13:04.51 [debug    ] Will speak dialect             dialect=http://json-
↪schema.org/draft-07/schema#
2022-10-31 13:04.52 [warning  ] Implicit dialect not acknowledged. Proceeding, but␣
↪implementation may not have configured itself to handle schemas without $schema.␣
↪[localhost/tutorial-lua-jsonschema] dialect=http://json-schema.org/draft-07/schema#␣
↪response=StartedDialect(ok=False)
{"implementations": {"localhost/tutorial-lua-jsonschema": {"language": "lua", "name":
↪"jsonschema", "homepage": "https://github.com/api7/jsonschema", "issues": "https://
↪github.com/api7/jsonschema/issues", "dialects": ["http://json-schema.org/draft-07/
↪schema#", "http://json-schema.org/draft-06/schema#", "http://json-schema.org/draft-04/
↪schema#"], "image": "localhost/tutorial-lua-jsonschema"}}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
───────────── localhost/tutorial-lua-jsonschema (stderr) ─────────────
│                                                                    │
│    luajit: bowtie_jsonschema.lua:42: attempt to call a nil value   │
│    stack traceback:                                                │
│            bowtie_jsonschema.lua:42: in main chunk                 │
│            [C]: at 0xaaaacecf8690                                  │
│                                                                    │
──────────────────────────────────────────────────────────────────────
2022-10-31 13:04.52 [error    ] uncaught error                 [localhost/tutorial-lua-
↪jsonschema] case=test case 1 schema={} seq=1
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 1, "context": {"stderr":
↪"luajit: bowtie_jsonschema.lua:42: attempt to call a nil value\nstack traceback:\n\
↪tbowtie_jsonschema.lua:42: in main chunk\n\t[C]: at 0xaaaacecf8690\n"}, "caught":␣
↪false}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
───────────── localhost/tutorial-lua-jsonschema (stderr) ─────────────
│                                                                    │
```

(continues on next page)

(continued from previous page)

```
│    luajit: bowtie_jsonschema.lua:42: attempt to call a nil value            │
│    stack traceback:                                                         │
│            bowtie_jsonschema.lua:42: in main chunk                          │
│            [C]: at 0xaaaae3078690                                           │
│                                                                             │
──────────────────────────────────────────────────────────────────────────────
2022-10-31 13:04.53 [error    ] uncaught error                 [localhost/tutorial-lua-
↪jsonschema] case=test case 2 schema={'const': 37} seq=2
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 2, "context": {"stderr":
↪"luajit: bowtie_jsonschema.lua:42: attempt to call a nil value\nstack traceback:\n\
↪tbowtie_jsonschema.lua:42: in main chunk\n\t[C]: at 0xaaaae3078690\n"}, "caught":␣
↪false}
2022-10-31 13:04.53 [info     ] Finished                       count=2
```

where we have one final command left to implement – actually running our test cases!

4.5 Step 3: Validating Instances

It's now time to actually invoke the implementation itself, so import the library by adding:

```
local jsonschema = require 'jsonschema'
```

to the top of your harness.

The API we need within the Lua `jsonschema` library is one called `jsonschema.generate_validator`, which is the API which a user of the library calls in order to validate an instance under a provided schema. For details on how to use this API, see the library's documentation, but for our purposes it essentially takes 2 arguments – a JSON Schema (represented as a Lua `table`) along with an additional options argument which we'll use momentarily. It returns a callable which then can be used to validate instances (other Lua values). Let's take a first pass at implementing the `run` command, whose input looks like:

```
{
  "description": "Sent to implementations for each test case.",

  "$id": "tag:bowtie.report,2023:ihop:command:run",

  "required": ["seq", "case"],
  "properties": {
    "cmd": { "const": "run" },
    "seq": { "$ref": "tag:bowtie.report,2024:report:seq" },
    "case": { "$ref": "tag:bowtie.report,2023:ihop#case" }
  },
```

`run` requests contain a test case (a schema with tests), alongside a `seq` parameter which is simply an identifier for the request and needs to be included in the response we write back. Here's an implementation of the `run` command to add to our harness implementation:

```
run = function(request)
  assert(STARTED, 'Not started!')

  local validate = jsonschema.generate_validator(request.case.schema)
```

(continues on next page)

(continued from previous page)

```
  local results = {}
  for _, test in ipairs(request.case.tests) do
    table.insert(results, { valid = validate(test.instance) })
  end
  return { seq = request.seq, results = results }
end,
```

We call `generate_validator` to get our validation callable, then we apply it (`map` it, though Lua has no builtin to do so) over all tests in the `run` request, returning a response which contains the `seq` number alongside results for each test. The results are indicated positionally as shown above, meaning the first result in the results array should be the result for the first test in the input array.

If we run `bowtie` again, we see:

```
2022-10-31 13:20.14 [debug    ] Will speak dialect          dialect=http://json-
↪schema.org/draft-07/schema#
2022-10-31 13:20.14 [warning  ] Implicit dialect not acknowledged. Proceeding, but␣
↪implementation may not have configured itself to handle schemas without $schema.␣
↪[localhost/tutorial-lua-jsonschema] dialect=http://json-schema.org/draft-07/schema#␣
↪response=StartedDialect(ok=False)
{"implementations": {"localhost/tutorial-lua-jsonschema": {"dialects": ["http://json-
↪schema.org/draft-07/schema#", "http://json-schema.org/draft-06/schema#", "http://json-
↪schema.org/draft-04/schema#"], "language": "lua", "name": "jsonschema", "homepage":
↪"https://github.com/api7/jsonschema", "issues": "https://github.com/api7/jsonschema/
↪issues", "image": "localhost/tutorial-lua-jsonschema"}}}
{"case": {"description": "test case 1", "schema": {}, "tests": [{"description": "a test",
↪ "instance": {}, "valid": null}], "comment": null, "registry": null}, "seq": 1}
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 1, "results": [{"valid":␣
↪true}], "expected": [null]}
{"case": {"description": "test case 2", "schema": {"const": 37}, "tests": [{"description
↪": "not 37", "instance": {}, "valid": null}, {"description": "is 37", "instance": 37,
↪"valid": null}], "comment": null, "registry": null}, "seq": 2}
{"implementation": "localhost/tutorial-lua-jsonschema", "seq": 2, "results": [{"valid":␣
↪false}, {"valid": true}], "expected": [null, null]}
2022-10-31 13:20.14 [info     ] Finished                    count=2
```

where we've now successfully run some inputted test cases. The output we see now contains the results returned by the Lua implementation and is ready to be piped into *bowtie summary*. Hooray!

4.6 Step 4: Resolving References

In order to support testing the $ref keyword from JSON Schema, which involves resolving references to JSON documents, there's an additional parameter that is sent with `run` commands which contains a schema *registry*, i.e. a collection of additional schemas beyond the test case schema itself which may be referenced from within the test case. The intention is that the harness should configure its implementation to be able to retrieve any of the schemas present in the registry for the duration of the test case.

As an example, a registry may look like:

```
{
  "http://example.com/my/string/schema": {"type": "string"}
}
```

which, if included in a `run` request means that the implementation is expected to resolve a retrieval URI of `http://example.com/my/string/schema` to the corresponding schema above. Each key in the registry is an (absolute) retrieval URI and each value is a corresponding JSON Schema.

The Lua implementation we have been writing a harness for actually contains no built-in support for resolving `$ref` by default, but does give us a way to "hook in" an implementation of reference resolution. Specifically, `generate_validator` takes a second argument (a Lua `table`) whose `external_resolver` key may point to a callable that is called when encountering a reference. If we pass the library a callable which simply retrieves references from the registry provided whenever encountering a `$ref`, we have implemented what's needed by Bowtie. Change the call to `generate_validator` to look like:

```
local validate = jsonschema.generate_validator(request.case.schema, {
  external_resolver = function(url)
    return request.case.registry[url]
  end,
})
```

where we simply index into `request.case.registry` anytime we see a referenced URL.

And *now* it would seem we're done. We could stop here, as we've done enough that Bowtie can take it from here, but there are a few finishing touches to implement which improve performance or overall experience interacting with our new harness, so let's get to those before we clean up shop.

4.7 Step 5: Handling Errors

If an implementation happens to not be fully compliant with the specification, or not fully compliant *yet*, Bowtie may end up passing a schema or instance that causes the harness to crash if the underlying implementation crashes (by panicking, raising an exception, etc. depending on the host language).

Whilst Bowtie tries to be hearty about these possibilities by automatically restarting crashed containers, it's more efficient for the harness itself to do so via mechanisms within the host language.

Bowtie will show an `uncaught error` message in its debugging output whenever a container crashes. We can make the harness internally catch the error(s) and return a special response to `run` requests which signals that the implementation errored. Doing so will still be marked as an error in debugging output, but Bowtie will recognize that the error was caught by the harness, and things will generally be faster by not incurring additional restart cost each time the harness crashes.

Catching exceptions from our Lua implementation is simple, by wrapping the `validate()` function call with the `pcall` function, which will catch any errors raised and allow us to detect whether any have occurred. Once the harness detects an error, it should return an error response (in place of results), which may include any diagnostic information for later use, e.g. a traceback or internal error message. The structure of error responses is:

```
        "errored": {
          "description": "Signal that the implementation encountered an internal error␣
↪(which it caught). Additional context can be passed along (e.g. a traceback or␣
↪exception detail).",

          "$anchor": "errored",

          "required": ["errored"],
          "properties": {
            "errored": { "const": true },
            "context": {
```

(continues on next page)

(continued from previous page)

```
              "description": "Additional implementation-specific or language-specific␣
↪context available when the error was caught.",
              "type": "object",
              "properties": {
                "message": {
                  "description": "A (short) description of the error.",
                  "type": "string"
                },
                "traceback": {
                  "description": "A language-specific traceback (backtrace) containing␣
↪stack frames or fuller debugging information on where the error occurred.",
                  "type": "string"
                },
                "stderr": {
                  "description": "The raw (captured) contents of standard error from␣
↪within the harness. Prefer using 'traceback' if the contents are a traceback.",
                  "type": "string"
                }
              },
              "additionalProperties": true
            }
          }
        }
      }
    }
  }
}
```

(i.e. in particular the harness should return a response setting `errored` to `true`).

4.8 Step 6: Skipping Tests & Handling Known Issues

The aforementioned error-handling support means that a running harness can be fairly sure it gracefully continues in the face of issues, even for unexpectedly new input.

But if an implementation has *known* gaps, it's better to explicitly tell Bowtie that a particular feature is unsupported, rather than simply letting the error handling code trap any exception or error. The reasoning here is that Bowtie can surface any detail about *why* a test is skipped, or some day may even notice when an issue on an implementation is closed (and retry running the test). In general, it is preferable that a test harness have 0 errors, and certainly 0 uncaught errors, when being run under the official suite (and instead intentionally skip known issues, while emitting a link to an issue tracking the missing functionality). Support for skipping tests is still somewhat crude, but it does indeed work, and should be preferred by an implementer if you know a particular test from the official suite isn't supported by your implementation.

The structure of skip responses, which you should send when presented with a "known" unsupported test, is:

```
"skipped": {
  "description": "Signal that the implementation skipped the test case or test␣
↪(typically because it is a known bug). Either an issue URL or a human-readable message␣
↪is encouraged to explain the skip.",
```

(continues on next page)

(continued from previous page)

```
  "$anchor": "skipped",

  "required": ["skipped"],
  "properties": {
    "skipped": { "const": true },
    "message": {
      "description": "A human-readable message passed back from the implementation␣
↪explaining why the skip occurred.",
      "type": "string"
    },
    "issue_url": {
      "description": "An optional link to a relevant issue on the implementation's bug␣
↪tracker.",

      "type": "string",
      "format": "url"
    }
  }
},
```

The biggest consideration at the minute is how to *identify* incoming test cases. Adding some sort of “persistent identifier” to test cases is something we’ve previously discussed upstream in the official test suite, which would make this easier. Until that happens however, your best current bet is to match on the test case description and/or schema, and use that to decide this incoming test is unsupported (and then respond with a skip request as above). For a specific example of doing so for Bowtie’s reporting, see this PR.

If you’ve gotten to the end and wish to see the full code for the harness, have a look at the completed harness for lua-jsonschema.

4.9 Addendum: Submitting Upstream

If the implementation you’ve added isn’t already supported by Bowtie, contributing it is very welcome!

Please feel free to open an issue or pull request adding it to the implementations directory.

If you do so there are a few additional things to do beyond the above:

- Commit your harness to the `implementations` directory in the root of Bowtie’s repository, alongside the existing ones. Name your harness directory `<host-language>-<implementation-name>/`. Use ASCII-compatible names, so if your implementation is written in C++ and is called `flooblekins` within the C++ ecosystem, call the directory `cpp-flooblekins/`.
- Ensure your harness is small and easily maintained, as it may need to be touched by others as Bowtie’s protocol changes.
- Install as little as possible from arbitrary URLs, preferring both the OS’s package manager and your language’s package manager whenever possible over manual download and compilation.
- Please ensure you’ve used an `alpine`-based image, or at least a slim one, to keep sizes as small as possible. Reference the existing `Dockerfiles` if you need inspiration.
- Please also add some sort of linter or autoformatter for the source code you’ve written. Because Bowtie has code from so many languages in it, it’s simply too much to expect any human eyes to catch style or formatting issues. Which autoformatter you use will depend on the host language, but if you look at the .pre-commit-config.yaml file in this repository you’ll see we run `go fmt` for Golang implementations, `cargo fmt` for Rust ones, `black`

for Python ones, etc., each being a "commonly used" tool in the corresponding language. If yours is the first implementation Bowtie supports in a language, use the "most commonly used" linter or autoformatter for the language.

- Finally, please enable dependabot support for your implementation whenever possible (or necessary). If your image does not pin the library being tested this may not be necessary (since each rebuild of the image should retrieve the latest version), but if it does (via e.g. a `package.json` in Javascript, a `requirements.txt` in Python, a `cargo.lock` for Rust, etc.) Dependabot can be used to automatically ensure we update Bowtie's implementation versions without manual labor.

CHAPTER

FIVE

INSTALLATION

Jump Right In!

You can use Bowtie immediately without installing it!

Both GitHub Codespaces and GitPod can provide you with an immediate, fully-working cloud environment with Bowtie and all of its supported implementations installed into it:

5.1 Via Homebrew (macOS or Linuxbrew)

Bowtie is available in a tap which is located here, and can be installed via:

```
brew install bowtie-json-schema/tap/bowtie
```

5.2 As a Single Executable

There is an experimental PyApp of Bowtie published to GitHub on each release.

You can find the latest one here.

Once downloading it, run `chmod +x` on it and you should be able to use it as-is if you have an existing Python installation.

If you use it (successfully or otherwise) please provide feedback.

5.3 Manual Installation via PyPI

Bowtie is written in Python, and uses a container runtime to execute JSON Schema implementations, so you'll need both Python and a suitable container runtime installed.

If you have no previous container runtime installed (e.g. Docker), follow the installation instructions for podman specific to your operating system. Ensure you've started a Podman VM if you are on macOS.

Then follow the pipx installation process to install `pipx`, and finally use it to install Bowtie via:

```
pipx install bowtie-json-schema
```

which should give you a working Bowtie installation, which you can check via:

```
bowtie --help
```

Further usage details of the command-line interface can be found *here*.

INDEX