

Programming OpenResty

OpenResty

Published
with GitBook



Table of Contents

Introduction	0
Automated Testing	1
Introduction	1.1
Test::Nginx	1.2
Test Suite Layout	1.3
Test File Layout	1.4
Running Tests	1.5
Preparing Tests	1.6
Testing Erroneous Cases	1.7
Test Modes	1.8
Advanced Topics	1.9

Programming OpenResty

This is an official guide on OpenResty programming written by the OpenResty creator. This book is still in preparation. Please check back often for updates.

The entire Programming OpenResty book, written by Yichun Zhang, is available here. All content is licensed under the [Creative Commons Attribution Non Commercial Share Alike 3.0 license](#). You can download or browse the rendered book in various different formats on the GitBook website below.

<https://www.gitbook.com/book/openresty/programming-openresty/>

The latest source of the book can be found in the following GitHub repository:

<https://github.com/openresty/programming-openresty>

Pull requests are always welcome.

Automated Testing

Automated testing plays a critical role in software development and maintainance. OpenResty provides a data-driven test scaffold for writing declarative test cases for NGINX C modules, Lua libraries, and even OpenResty applications. The test cases are written in a specification-like format, which is both intuitive to read and write for humans and also easy to handle for machines. The data-driven approach makes it easy to run the same tests in wildly different ways that can help expose issues in different scenarios or with different kinds of external tools.

This chapter introduces the `Test::Nginx` test scaffold that has been widely used to organize test suites for almost all the OpenResty components, including the `ngx_http_lua` module, most of the `lua-resty-*` Lua libraries, as well as full-blown business applications like CloudFlare's Lua CDN and Lua SSL.

Keywords: Testing, Mocking

Introduction

OpenResty itself has been relying on automated testing to remain high quality over the years. As OpenResty core developers, we embrace the test driven development (TDD) process all the time. An excellent result of our TDD practices over the years is a huge set of test suites for all the OpenResty components. These test suites are so large as a whole, so it is impractical to run all the tests thoroughly on a single machine. A relatively large test cluster is often run on Amazon EC2 to run all these tests in all existing test modes. Lying at the heart of these test suites is usually the `Test::Nginx` test scaffold module developed by the OpenResty team.

The `Test::Nginx` scaffold provides a generic simple specification language for expressing and organizing test cases in an intuitive way. It also provides various powerful testing modes or "engines" to run the tests in various different ways in the hope of exposing bugs in different settings. It is also supported to extend the test specification language to add custom abstractions for advanced testing needs, usually found in application-level regression testing.

Conceptual Roadmap

Overview

Test::Nginx

`Test::Nginx` is a test framework that drives test cases written for any code running atop NGINX, and also, naturally, the NGINX core itself. It is written in Perl because of the rich testing facilities and toolchain already accumulated in the Perl world for years. Fortunately, the user does not really need to know Perl for writing test cases atop this scaffold since `Test::Nginx` provides a very simple notation to present the test cases in a specification-like format.

The simple test specification format, or language, used in `Test::Nginx` is just a dialect of the more general testing language provided by the `Test::Base` testing module in the Perl world. In fact, `Test::Nginx` is just a subclass of `Test::Base` in the sense of object-oriented programming. This means that all the features offered by `Test::Base` is available in `Test::Nginx` and `Test::Nginx` just provides handy primitives and notations that simplify testing in the NGINX and OpenResty context. The core idea of `Test::Base` is so useful that we have been using testing scaffolds based on `Test::Base` in many different projects even including Haskell programs and Linux kernel modules. `Test::Nginx` is such an example we created for the NGINX and OpenResty world. Detailed discussion of the `Test::Base` framework itself is beyond the scope of this book, but we will introduce the important features of `Test::Base` that are inherited by `Test::Nginx` in the later sections.

`Test::Nginx` is distributed via [CPAN](#), the Comprehensive Perl Archive Network, just like most of the other Perl libraries. If you already have `perl` installed in your system (many Linux distributions ship with `perl` by default), then you can install `Test::Nginx` with the following simple command:

```
cpan Test::Nginx
```

For the first time that the `cpan` utility is run, you may be prompted to configure the `cpan` utility to fit your requirements. If you are unsure about those options, just choose the automatic configuration option (if available) or just accept all the default settings.

`Test::Nginx` provides several different testing classes for different user requirements. The most frequently used one is `Test::Nginx::Socket`. The rest of this chapter will focus on this testing class and its subclasses. We will use the names `Test::Nginx` and `Test::Nginx::Socket` interchangeably from now on to mean the `Test::Nginx::Socket` test module and its subclasses, unless otherwise specified.

Note

There is actually another different testing scaffold called `Test::Nginx`, created by Maxim Dounin and maintained by the official NGINX team. That testing module is shipped with the [official NGINX test suite](#) and has no relationship with our `Test::Nginx` except that both of these are meant to test NGINX related code. The NGINX team's `Test::Nginx` requires the user to directly code in Perl to convey all the test cases, which means that tests written for their `Test::Nginx` are not data driven and requires decent knowledge about Perl programming.

Test Suite Layout

Projects using `Test::Nginx` to drive their test suites usually have a common directory layout and common test file name patterns to organize their tests. This makes the user easy to reason about the location of the test suite in a project source tree and the usage of the tests. It is not really required, however, to use this common convention; it is just highly recommended.

By convention, such projects have a `t/` directory at the root of their source tree where test files reside in. Each test file contains test cases that are closely related in some way and has the file extension `.t` to easily identify themselves as "test files". Below is the directory tree structure of a real-world test suite inside the [headers-more-nginx-module](#) project:

```
└─ t
   ├── bug.t
   ├── builtin.t
   ├── eval.t
   ├── input-conn.t
   ├── input-cookie.t
   ├── input-ua.t
   ├── input.t
   ├── phase.t
   ├── sanity.t
   ├── subrequest.t
   ├── unused.t
   └─ vars.t
```

When you have many test files, you can also group them further with sub-directories under `t/`. For example, in the [lua-nginx-module](#) project, we have sub-directories like `023-rewrite/` and `024-access/` under its `t/` directory.

In essence, each `.t` file is a Perl script file runnable by either `perl` or Perl's universal test harness tool named [prove](#). We usually use the `prove` command-line utility to run such `.t` files to obtain test results. Although `.t` files are Perl scripts per se, they usually do not have much Perl code at all. Instead, all of the test cases are declared as cleanly formatted "data" in these `.t` files.

Note

The test suite layout convention we use here are also used by the Perl community for many years. Because `Test::Nginx` is written in Perl and reuses Perl's testing toolchain, it makes sense for us to simply follow that convention in the NGINX and OpenResty world as well.

Test File Layout

Test files usually have a common file extension, `.t`, to distinguish themselves from other types of files in the source tree. Each test file is a Perl script per se. `Test::Nginx` follows a special design that decomposes each test file into two main parts: the first part is a very short prologue that consists of a few lines of Perl code while the second part is a listing of the test cases in a special data format. These two parts are separated by the following special line

```
__DATA__
```

The `perl` interpreter or the `prove` utility stop interpreting the file content as Perl source code until they see this special line. Everything after this line is treated as **data** in plain text that is reachable by the Perl code above this line. The most interesting part of each `.t` test file is the stuff after this line, i.e., the data part.

Note	<p>The special <code>__DATA__</code> notation is a powerful feature of the Perl programming language that allows embedding arbitrary free-text data in any Perl script files that can be manipulated by the containing Perl scripts themselves.</p> <p><code>Test::Nginx</code> takes advantage of this feature to allow data-driven test case specifications in a simple format or language that is easily understandable by everyone, even those without any prior experiences in Perl programming.</p>
------	---

The Prologue Part

The first part, i.e., the "prologue" above the `__DATA__` line is usually just a few lines of Perl code. You do not have to know Perl programming to write them down because they are so simple and seldom or never change. The simplest Perl code prologue is as follows:

```
use Test::Nginx::Socket 'no_plan';
run_tests();
```

The first line is just loading the Perl module (or class), `Test::Nginx::Socket` and passing the option `'no_plan'` to it to disable test plans (we will talk more about test plans in later chapters and we do not bother worrying about it here). `Test::Nginx::Socket` is one of the most popular class in the `Test::Nginx` test framework. The second line just calls the `run_tests` Perl function imported automatically from the `Test::Nginx::Socket` module to run all the test cases defined in the data part of the test file (i.e., the things coming after the `__DATA__` line).

There are, however, more complicated prologue parts in many real-world test suites. Such prologues usually define some special environment variables or Perl variables that can be shared and referenced in the test cases defined in the "data part", or just call some other Perl functions imported by the `Test::Nginx::Socket` module to customize the testing configurations and behaviors for the current test file. We will return to such fancier prologues in later sections. They can be very helpful in some cases.

Note

Perl allows function calls to omit the parentheses if the context is unambiguous. So we may see Perl function calls without parentheses in real-world test files' prologue part, like `run_tests;`. We may use such forms in examples presented in later sections because they are more compact.

The Data Part

The data part is the most important part of any test files powered by `Test::Nginx`. This is where test cases reside. It uses a simple specification format to express test cases so that the user does not use Perl or any other general-purpose languages to present the tests themselves. This special specification format is an instance of Domain-Specific Languages (DSL) where the "domain" is defined as testing code running upon or inside NGINX. Use of a DSL to present test cases open the door of presenting the test cases as *data* instead of code. This is also why `Test::Nginx` is a data-driven testing framework.

The test case specification in the data part is composed by a series of *test blocks*. Each test block usually corresponds to a single test case, which has a *title*, an optional *description*, and a series of *data sections*. The structure of a test block is described by the following template.

```
=== title
optional description
goes here...
--- section1
value1 goes
here
--- section2
value2 is
here
--- section3
value3
```

Block Titles

As we can see, each test block starts with a title line prefixed by three equal sign (`===`). It is important to *avoid* any leading spaces at the beginning of the line. The title is mandatory and is important to describe the intention of the current test case in the most concise form, and

also to identify the test block in the test report when test failures happen. By convention we put a `TEST N:` prefix in this title, for instance, `TEST 3: test the simplest form`. Don't worry about maintaining the test ordinal numbers in these titles yourself, we will introduce a command-line utility called [reindex](#) in a later section that can automatically update the ordinal numbers in the block titles for you.

Block Descriptions

Each test block can carry an optional description right after the block title line. This description can span multiple lines if needed. It is a more detailed description of the intention of the test block than the block title and may also give some background information about the current test. Many test cases just omit this part for convenience.

Data Sections

Every test block carries one or more *data sections* right after the block description (if any). Data sections always have a name and a value, which specify any input data fields and the *expected* output data fields.

The name of a data section is the word after the line prefix `---`. Spaces are allowed though not syntactically required after `---`. We usually use a single space between the prefix and the section name for aesthetic considerations and we hope that you follow this convention as well. The section names usually contain just alphanumeric letters and underscore characters.

Section values are specified in two forms. One is all the lines after the section name line, before the next section or the next block. The other form is more concise and specifies the value directly on the same line as the section name, but right after the first colon character (`:`). The latter form requires that the value contains no line-breaks. Any spaces around the colon are always discarded and never count as a part of the section value; furthermore, the trailing line-break character in the one-line form does not count either.

If no visible values come after the section name in either form, then the section takes an empty string value, which is still a *defined* value, however. On the other hand, omitting the section name (and value) altogether makes that section *undefined*.

`Test::Nginx` offers various pre-defined data section names that can be used in the test blocks for different purposes. Some data sections are for specifying input data, some are for expected output, and some for controlling whether the current test block should be run at all.

It is best to explain data sections in a concrete test block example.

```
=== TEST 1: hello, world
This is just a simple demonstration of the
echo directive provided by ngx_http_echo_module.
--- config
location = /t {
    echo "hello, world!";
}
--- request
GET /t
--- response_body
hello, world!
--- error_code: 200
```

Here we have two input data sections, `config` and `request`, for specifying a custom NGINX configuration snippet in the default `server {}` and the HTTP request sent by the test scaffold to the test NGINX server, respectively. In addition, we have one output data section, `response_body`, for specifying the expected response body output by the test NGINX server. If the actual response body data is different from what we specify under the `response_body` section, this test case fails. We have another output data section, `error_code`, which specifies its value on the same line of the section name. We see that a colon character is used to separate the section name and values. Obviously, the `error_code` section specifies the expected HTTP response status code, which is 200.

Empty lines around data sections are always discarded by `Test::Nginx::Socket`. Thus the test block above can be rewritten as below without changing its meaning.

```
=== TEST 1: hello, world
This is just a simple demonstration of the
echo directive provided by ngx_http_echo_module.

--- config
location = /t {
    echo "hello, world!";
}

--- request
GET /t

--- response_body
hello, world!

--- error_code: 200
```

Some users prefer this style for aesthetic reasons. We are free to choose whatever form you like.

There are also some special data sections that specify neither input nor output. They are just used to *control* how test blocks are run. For example, the `ONLY` section makes *only* the current test block in the current test file run and all the other test blocks are skipped. This is extremely useful for running an individual test block in any given file, which is a common requirement while debugging a particular test failure. Also, the special `SKIP` section can skip running the containing test block unconditionally, handy for preparing test cases for future features without introducing any expected test failures. We will visit more such "control sections" in later sections.

We shall see, in a later section, that the user can define her own data sections or extending existing ones by writing a little bit of custom Perl code to satisfy her more complicated testing requirements.

Section Filters

Data sections can take one or more *filters*. Filters are handy when you want to adjust or convert the section values in certain ways.

Syntactically, filters are specified right after the section name with at least one space character as the separator. Multiple filters are also separated by spaces and are applied in the order they are written.

`Test::Nginx::Socket` provides many filters for your convenience. Consider the following data section from the aforementioned test block.

```
--- error_code: 200
```

If we want to place the section value, 200, in a separate line, like below,

```
--- error_code
200
```

then the section value would contain a trailing new line, which leads to a test failure. This is because the one-line form always excludes the trailing new-line character while the multi-line form always includes one. To explicitly exclude the trailing new-line in the multi-line form, we can employ the `chomp` filter, as in

```
--- error_code chomp
200
```

Now it has exactly the same semantics as the previous one-line form.

Some filters have more dramatic effect on the section values. For instance, the `eval` filter evaluates the section value as arbitrary Perl code, and the Perl value resulted from the execution will be used as the final section value. The following section demonstrates using the `eval` filter to produce 4096 a's:

```
--- response_body eval
"a" x 4096
```

The original value of the `response_body` section above is a Perl expression where the `x` symbol is a Perl operator is used to construct a string that repeats the string specified as the left-hand-side N times where N is specified by the right-hand-side. The resulting 4096-byte Perl string after evaluating this expression dictated by the `eval` filter will be used as the final section value for comparison with the actual response body data. It is obvious that use of the `eval` filter and a Perl expression here is much more readable and manageable by directly pasting that 4096-byte string in the test block.

As with data sections, the user can also define her own filters, as we shall see in a later section.

A Complete Example

We can conclude this section by a complete test file example given below, with both the prologue part and the data part.

```
use Test::Nginx::Socket 'no_plan';

run_tests();

__DATA__

=== TEST 1: hello, world
This is just a simple demonstration of the
echo directive provided by ngx_http_echo_module.
--- config
location = /t {
    echo "hello, world!";
}
--- request
GET /t
--- response_body
hello, world!
--- error_code: 200
```

We will see how to actually run such test files in the next section.

Note

The test file layout described in this section is exactly the same as the test files based on other test frameworks derived from `Test::Base`, the superclass of `Test::Nginx::Socket`, except those specialized test sections and specialized Perl functions defined only in `Test::Nginx::Socket`. All the `Test::Base` derivatives share the same basic layout and syntax. They proudly inherit the same veins of blood.

Running Tests

Like most Perl-based testing frameworks, `Test::Nginx` relies on Perl's `prove` command-line utility to run the test files. The `prove` utility is usually shipped with the standard perl distribution so we should already have it when we have `perl` installed.

`Test::Nginx` always invokes a real NGINX server and a real socket client to run the tests. It automatically uses the `nginx` program found in the system environment `PATH`. It is your responsibility to specify the right `nginx` in your `PATH` environment for the test suite. Usually we just specify the path of the `nginx` program inside the `OpenResty` installation tree. For example,

```
export PATH=/usr/local/openresty/nginx/sbin:$PATH
```

Here we assume that OpenResty is installed to the default prefix, i.e.,

```
/usr/local/openresty/.
```

You can always use the `which` command to verify if the `PATH` environment is indeed set properly:

```
$ which nginx
/usr/local/openresty/nginx/sbin/nginx
```

For convenience, we usually wrap such environment settings in a custom shell script so that we do not risk polluting the system-wide or account-wide environment settings nor take on the burden of manually setting the environments manually for every shell session. For example, I usually have a local bash script named `go` in each project I work on. A typical `go` script might look like below

```
#!/usr/bin/env bash

export PATH=/usr/local/openresty/nginx/sbin:$PATH

exec prove "$@"
```

Then we can use this `./go` script to substitute the `prove` utility in any of the subsequent commands involving `prove`.

Because `Test::Nginx` makes heavy use of environment variables for the callers to fine tune the testing behaviors (as we shall see in later sections), such shell wrapper scripts also make it easy to manage all these environment variable settings and hard to get things

wrong.

Note

Please do not confuse the name of this bash script with Google's Go programming language. It has nothing to do with the Go language in any way.

Running A Single File

If you want to run a single test file, say, `t/foo.t`, then all you need to do is just typing the following command in your terminal.

```
prove t/foo.t
```

Here inside `t/foo.t` we employs the simple test file example presented in the previous section. We repeat the content below for the reader's convenience.

`t/foo.t`

```
use Test::Nginx::Socket 'no_plan';

run_tests();

__DATA__

=== TEST 1: hello, world
This is just a simple demonstration of the
echo directive provided by ngx_http_echo_module.
--- config
location = /t {
    echo "hello, world!";
}
--- request
GET /t
--- response_body
hello, world!
--- error_code: 200
```

It is worth mentioning that we could run the following command instead if we have a custom wrapper script called `./go` for `prove` (as mentioned earlier in this section):

```
./go foo.t
```

When everything goes well, it generates an output like this:

```
t/foo.t .. ok
All tests successful.
Files=1, Tests=2, 0 wallclock secs (0.02 usr 0.01 sys + 0.08 cusr 0.00 csys)
Result: PASS
```

This is a very concise summary. The first line tells you all tests were passed while the second line gives you a summary of the number of test files (1 in this case), the number of tests (2 in this case), and the wallclock and CPU times used to run all the tests.

It is interesting to see that we have only one test block in the sample test file but in the test summary output by `prove` we see that the number of tests are 2. Why the difference? We can easily find it out by asking `prove` to generate a detailed test report for all the individual tests. This is achieved by passing the `-v` option (meaning "verbose") to the `prove` command we used earlier:

```
prove -v t/foo.t
```

Now the output shows all the individual tests performed in that test file:

```
t/foo.t ..
ok 1 - TEST 1: hello, world - status code ok
ok 2 - TEST 1: hello, world - response_body - response is expected
1..2
ok
All tests successful.
Files=1, Tests=2, 0 wallclock secs (0.01 usr 0.01 sys + 0.07 cusr 0.00 csys)
Result: PASS
```

Obviously, the first test is doing the status code check, which is dictated by the `error_code` data section in the test block, and the second test is doing the response body check, required by the `response_body` section. Now the mystery is solved.

It is worth mentioning that the `--- error_code: 200` section is automatically assumed when no `error_code` section is explicitly provided in the test block. So our test block above can be simplified by removing the `--- error_code: 200` line without affecting the number of tests. This is because that checking 200 response status code is so common that `Test::Nginx` makes it the default. If you expect a different status code, like 500, then just add an explicit `error_code` section.

From this example, we can see that one test block can contain multiple tests and the number of tests for any given test block can be determined or predicted by looking at the data sections performing output checks. This is important when we provide a "test plan" ourselves to the test file where a "test plan" is the exact number of tests we *expect* the current test file to run. If a different number of tests than the plan were actually run, then the test result would be considered malicious even when all the tests are passed successfully. Thus, a test plan adds a strong constraint on the total number of tests expected to be run. For our `t/foo.t` file here, however, we intentionally avoid providing any test plans by passing the `'no_plan'` argument to the `use` statement that loads the `Test::Nginx::Socket` module. We will revisit the "test plan" feature and explain how to provide one in a later section.

Running Multiple Files

Running multiple test files are straightforward; just specify the file names on the `prove` command line, as in

```
prove -v t/foo.t t/bar.t t/baz.t
```

If you want to run all the test files directly under the `t/` directory, then using a shell wildcard can be handy:

```
prove -v t/*.t
```

In case that you have sub-directories under `t/`, you can specify the `-r` option to ask `prove` to recursively traverse the whole directory tree rooted at `t/` to find test files:

```
prove -r t/
```

This command is also the standard way to run the whole test suite of a project.

Running Individual Test Blocks

`Test::Nginx` makes it easy to run an individual test block in a given file. Just add the special data section `ONLY` to that test block you want to run individually and `prove` will skip all the other test blocks while running that test file. For example,

```

=== TEST 1: hello, world
This is just a simple demonstration of the
echo directive provided by ngx_http_echo_module.
--- config
location = /t {
    echo "hello, world!";
}
--- request
GET /t
--- response_body
hello, world!
--- ONLY

```

Now `prove` won't run any other test blocks (if any) in the same test file.

This is very handy while debugging a particular test block. You can focus on one test case at a time without worrying about other unrelated test cases stepping in your way.

When using the [Vim](#) editor, we can quickly insert a `--- ONLY` line to the test block we are viewing in the vim file buffer, and then type `!prove %` in the command mode of vim without leaving the editor window. This works because vim automatically expands the special `%` placeholder with the path of the current active file being edited. This workflow is great since you never leave your editor window and you never have to type the title (or other IDs) of your test block nor the path of the containing test file. You can quickly jump between test blocks even across different files. Test-driven development usually demands very frequent interactions and iterations, and `Test::Nginx` is particularly optimized to speed up this process.

Sometimes you may forget to remove the `--- ONLY` line from some test files even after debugging, this will incorrectly skip all the other tests in those files. To catch such mistakes, `Test::Nginx` always reports a warning for files using the `ONLY` special section, as in

```

$ prove t/foo.t
t/foo.t .. # I found ONLY: maybe you're debugging?
t/foo.t .. ok
All tests successful.
Files=1, Tests=2, 0 wallclock secs (0.01 usr 0.00 sys + 0.09 cusr 0.03 csys = 0.13 CPU)
Result: PASS

```

This way it is much easier to identify any leftover `--- ONLY` lines.

Similar to `ONLY`, `Test::Nginx` also provides the `LAST` data section to make the containing test block become the last test block being run in that test file.

Note	The special data sections <code>ONLY</code> and <code>LAST</code> are actually features inherited from the <code>Test::Base</code> module.
------	--

Skipping Tests

We can specify the special `SKIP` data section to skip running the containing test block unconditionally. This is handy when we write a test case that is for a future feature or a test case for a known bug that we haven't had the time to fix right now.

It is also possible to skip a whole test file in the prologue part. Just replace the `use` statement with the following form.

```
use Test::Nginx::Socket skip_all => "some reasons";
```

Then running the test file gives something like follows.

```
t/foo.t .. skipped: some reasons
```

Note	It is also possible to conditionally skip a whole test file but it requires a little bit of Perl programming. Interested readers can try using a <code>BEGIN {}</code> before before the <code>use</code> statement to calculate the value of the <code>skip_all</code> option on the fly.
------	--

Test Running Order

Test File Running Order

Test files are usually run by the alphabetical order of their file names. Some people prefer explicitly controlling the running order of their test files by prefixing the test file names with number sequences like `001-`, `002-`, and etc.

The test suite of the [ngx_http_lua](#) module follows this practice, for example, which has test file names like below

```
t/000-sanity.t
t/001-set.t
t/002-content.t
t/003-errors.t
...
t/139-ssl-cert-by.t
```

Although the `prove` utility supports running test files in multiple parallel jobs via the `-jN` option, `Test::Nginx` does not really support this mode since all the test cases share exactly the same test server directory, `t/servroot/`, and the same listening ports, as we have already seen, while parallel running requires strictly isolated running environments for each individual thread of execution. One can still manually split the test files into different groups and run each group on a different (virtual) machine or an isolated environment like a Linux container.

Test Block Running Order

By default, the `Test::Nginx` scaffold *shuffles* the test blocks in each file and run them in a *random* order. This behavior encourages writing self-contained and independent test cases and also increases the chance of hitting a bug by actively mutating the relative running order of the test cases. This may, indeed, confuse new comers, coming from a more traditional testing platform.

We can always disable this test block shuffling behavior by calling the Perl function, `no_shuffle()`, imported by the `Test::Nginx::Socket` module, before the `run_tests()` call in the test file prologue. For example,

```
use Test::Nginx::Socket 'no_plan';

no_shuffle();
run_tests();

__DATA__
...
```

With the `no_shuffle()` call in place, the test blocks are run in the exact same order as their appearance in the test file.

Preparing Tests

As we have seen in the previous sections, `Test::Nginx` provides a simple declarative format to express test cases. Each test case is represented by a test block. A test block consists of a title, an optional description, and several data sections for specifying inputs and expected outputs. In this section, we will have a close look at how to prepare such test cases for different test requirements.

Designing test cases is an art, in many ways. It may, sometimes, take even more time and effort than implementing the feature to be tested, according to our own experience.

`Test::Nginx` tries hard to make writing tests as simple as possible but it still cannot automate the whole test case design process. Only you know exactly what to test and how it can be tested anyway. This section will focus on the basic primitives provided by

`Test::Nginx` that you can take advantage of to devise clever and effective test cases.

Preparing NGINX Configuration

In a test block, we can use different data sections to specify our custom snippets in different positions of the final `nginx.conf` configuration file generated by `Test::Nginx`.

The most common one is the `config` section which is used to insert custom snippets inside the `server {}` configuration block for the default test server. We can also use the `http_config` section to insert our custom content into the `http {}` configuration block of `nginx.conf`. The `main_config` section can be used to insert content into the top-level scope of the NGINX configuration. Let's consider the following example.


```
=== TEST 1:
--- main_config
env MY_ENVIRONMENT;

--- http_config
    init_worker_by_lua_block {
        print("init")
    }

--- config
    location = /t {
        echo ok;
    }

--- request
GET /t
--- response_body
ok
```

This test block will generate an `nginx.conf` file with the following basic structure:

```
...
env MY_ENVIRONMENT;

http {
    ...

    init_worker_by_lua_block {
        print("init")
    }

    server {
        ...

        location = /t {
            echo ok;
        }
    }
}
```

Please pay attention to how the `main_config`, `http_config`, and `config` data sections' values are mapped into different locations in the NGINX configuration file.

When in doubt, we can always check out the actual `nginx.conf` file generated by the test scaffold at the location `t/servroot/conf/nginx.conf` in the current working directory (usually just being the root directory of the current project).

`Test::Nginx` generates a new `nginx.conf` file for each test block, which makes it possible for each test block to become self-contained. By default, the test scaffold automatically starts a new NGINX server before running each test block and shuts down the server immediately after running the block. Fortunately, NGINX is a lightweight server and it is usually very fast to start and stop. Thus, the test blocks are not that slow to run as it might look.

Preparing Requests

The simplest way to prepare a request is to use the `request` data section, as in

```
--- request
GET /t?a=1&b=2
```

The HTTP/1.1 protocol is used by default. You can explicitly make it use the HTTP/1.0 protocol if desired:

```
--- request
GET /t?a=1&b=2 HTTP/1.0
```

Leading spaces or empty lines in the value of the `request` section are automatically discarded. You can even add comments by leading them with a `#` character, as in

```
--- request

# this is a simple test:
GET /t
```

You can add some additional request headers at the same time through the `more_headers` section as below.

```
--- request
GET /t
--- more_headers
Foo: bar
Bar: baz
```

Pipelined Requests

Preparing pipelined HTTP requests are also possible. But you need to use the `pipelined_requests` section instead of `request`. For instance,

```
=== TEST 1: pipelined requests
--- config
    location = /t {
        echo ok;
    }

--- pipelined_requests eval
["GET /t", "GET /t"]

--- response_body eval
["ok\n", "ok\n"]
```

It is worth noting that we use the `eval` filter with the `pipelined_requests` section to treat the literal value of that section as Perl code. This way we can construct a Perl array of the request strings, which is the expected data format for the `pipelined_requests` section. Similarly we need a similar trick for the `response_body` section when checking outputs. With an array of expected response body data, we can expect and check different values for different individual request in the pipeline. Note, however, not every data section supports the same array-typed value semantics as `response_body`.

Checking Responses

We have already visited the `response_body` and `error_code` data sections for checking the response body data and response status code, respectively.

The `response_body` data section always performs an exact whole-string comparison between the section value and the actual response body. It tries to be clever when long string value comparison fails. Consider the following sample output from `prove`.

```
t/foo.t .. 1/?
#   Failed test 'TEST 1: long string test - response_body - response'
#   at .../test-nginx/lib/Test/Nginx/Socket.pm line 1282.
#       got: ..."IT 2.x is enabled.\x{0a}\x{0a}"...
#       length: 409
#   expected: ..."IT 2.x is not enabled.\x{0a}"...
#       length: 412
#   strings begin to differ at char 400 (line 1 column 400)
# Looks like you failed 1 test of 2.
/tmp/foo.t .. Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/2 subtests

Test Summary Report
-----
/tmp/foo.t (Wstat: 256 Tests: 2 Failed: 1)
  Failed test:  2
  Non-zero exit status: 1
Files=1, Tests=2,  0 wallclock secs (0.01 usr 0.00 sys + 0.09 cusr
Result: FAIL
```

From this test report, we can clearly see that

1. it is the test block with the title `TEST 1: long string test` that is failing,
2. it is the `response_body` data section check that fails,
3. the actual response body data is 409 bytes long while the expected value is 412 bytes, and
4. the expected value has an additional `not` word in the string fragment `IT 2.x is enabled` and the difference starts at the offset 400 in the long string.

Behind the scene, `Test::Nginx` uses the Perl module [Test::LongString](#) to do the long string comparisons. It is also particularly useful while checking response body data in binary formats.

If your response body data is in a multi-line textual format, then you may also want to use a `diff`-style output when the data does not match. To achieve this, we can call the `no_long_string()` Perl function before the `run_tests()` function call in the prologue part of the test file. Below is such an example.

```
use Test::Nginx::Socket 'no_plan';

no_long_string();

run_tests();

__DATA__

=== TEST 1:
--- config
    location = /t {
        echo "Life is short.";
        echo "Moon is bright.";
        echo "Sun is shining.";
    }
--- request
GET /t
--- response_body
Life is short.
Moon is deem.
Sun is shining.
```

Note the `no_long_string()` call in the prologue part. It is important to place it before the `run_tests()` call otherwise it would be too late for it to take effect, obviously.

Invoking the `prove` utility (or any shell wrappers for it) to run this test file gives the following details about the test failure:

```
# Failed test 'TEST 1: - response_body - response is expected (re
# at ../test-nginx/lib/Test/Nginx/Socket.pm line 1277.
# @@ -1,3 +1,3 @@
# Life is short.
# -Moon is deem.
# +Moon is bright.
# Sun is shining.
# Looks like you failed 1 test of 2.
```

It is obvious that the second line of the response body output is different.

You can even further disable the `diff` -style comparison mode by adding a `no_diff()` Perl function call in the prologue part. Then the failure report will look like this:

```
# Failed test 'TEST 1: - response_body - response is expected (re
# at .../test-nginx/lib/Test/Nginx/Socket.pm line 1277.
#      got: 'Life is short.
# Moon is bright.
# Sun is shining.
# '
#      expected: 'Life is short.
# Moon is deem.
# Sun is shining.
# '
# Looks like you failed 1 test of 2.
```

That is, `Test::Nginx` just gives full listing of the actual response body data and the expected one without any abbreviations or hand-holding.

Pattern Matching on Response Bodies

When the request body may change in some ways or you just care about certain key words in a long data string, you can specify a Perl regular expression to do a pattern match against the actual request body data. This is achieved by the `response_body_like` data section. For example,

```
--- response_body_like: age: \d+
```

Be careful when you are using the multi-line data section value form. A trailing newline character appended to your section value may make your pattern never match. In this case the `chomp` filter we introduced in an early section can be very helpful here. For example,

```
--- response_body_like chomp
age: \d+
```

You can also use the `eval` filter to construct a Perl regular expression object with a Perl expression, as in

```
--- response_body_like eval
qr/age: \d+/
```

This is the most flexible form to specify a pattern.

Note

Perl uses the `qr` quoting structure to explicitly construct regular expression objects. You can use various different quoting forms like `qr/.../` , `qr!...!` , `qr#...#` , and `qr{...}` .

Checking Response Headers

The `response_headers` data section can be used to validate response header entries. For example,

```
--- response_headers
Foo: bar
Bar: baz
!Blah
```

This section dictates 3 tests actually:

1. The response header `Foo` must appear and must take the value `bar` ;
2. The response header `Bar` must appear and must take the value `baz` ; and
3. The response header `Blah` must not appear or take an empty value.

Checking NGINX Error Logs

In addition to responses, the NGINX error log file is also an important output channel for an NGINX server setup.

True-False Tests

One immediate testing requirement is to check whether or not a piece of text appears in any error log messages. Such checks can be done via the data sections `error_log` and `no_error_log` , respectively. The former ensures that some lines in the error log file contain the string specified as the section value while the latter tests the opposite: ensuring that no line contains the pattern.

For example,

```
--- error_log
Hello world from my server
```

Then the string `Hello world from my server` (without the trailing new-line) must appear in at least one line of the NGINX error log. You can specify multiple strings in separate lines of the section value to perform different checks, for instance,

```

--- error_log
This is a dog!
Is it a cat?

```

Then it performs two error log checks, one is to ensure that the string `This is a dog!` appears in some error log lines. The order of these two string patterns do not matter at all.

If one of the string pattern failed to match any lines in the error log file, then we would get a test failure report from `prove` like below.

```
# Failed test 'TEST 1: simple test - pattern "This is a dog!" match
```



If you want to specify a Perl regular expression (regex) as one of the patterns, then you should use the `eval` section filter to construct a Perl-array as the section value, as in

```

--- error_log eval
[
  "This is a dog!",
  qr/\w+ is a cat\?/,
]

```

As we have seen earlier, Perl regexes can be constructed via the `qr/.../` quoting syntax. Perl string patterns in the Perl array specified by double quotes or single quotes are still treated as plain string patterns, as usual. If the array contains only one regex pattern, then you can omit the array itself, as in

```

--- error_log eval
qr/\w+ is a cat\?/

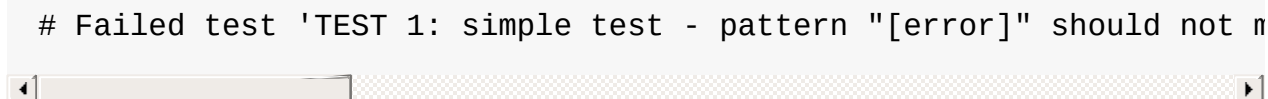
```

`Test::Nginx` puts the error log file of the test NGINX server in the file path `t/servroot/logs/error.log`. As a test writer, we frequently check out this file directly when things go wrong. For example, it is common to make mistakes or typos in the patterns we specify for the `error_log` section. Also, scanning the raw log file can give us insight about the details of the NGINX internal working when the NGINX debugging logs are enabled in the NGINX build.

The `no_error_log` section is very similar to `error_log` but it checks the nonexistence of the string patterns in the NGINX error log file. One of the most frequent uses of the `no_error_log` section is to ensure that there is *no* error level messages in the log file.


```
--- no_error_log
[error]
```

If, however, there is a line in the nginx error log file that contains the string `[error]`, then the test fails. Below is such an example.



```
# Failed test 'TEST 1: simple test - pattern "[error]" should not m
```

This is a great way to find the details of the error quickly by just looking at the test report.

Like `error_log`, this section also supports Perl array values and Perl regex values though the `eval` filter.

Grep Tests

The `error_log` and `no_error_log` sections are very handy in quickly checking the appearance of contain patterns in the NGINX error log file. But they have serious limitations in that it is impossible to impose stronger constraints on the relative order of the messages containing the patterns nor on the number of their occurrences.

To address such limitations, `Test::Nginx::Socket` provides an alternative way to check NGINX error logs in a way similar to the famous UNIX tool, `grep`. The sections `grep_error_log` and `grep_error_log_out` are used for this purpose. The test writer uses the `grep_error_log` section to specify a pattern, with which the test framework scans through the NGINX error log file and collect all the matched parts of the log file lines along the way, forming a final result. This aggregated log data result is then matched against the expected value specified as the value of the `grep_error_log_out` section, in a similar way as with the `response_body` section discussed above.

It is easiest to explain with a simple example.

```

=== TEST 1: simple grep test for error logs
--- config
    location = /t {
        content_by_lua_block {
            print("it is matched!")
            print("it is matched!")
            print("it is matched!")
        }
    }
--- request
GET /t
--- grep_error_log: it is matched!
--- grep_error_log_out
it is matched!
it is matched!
it is matched!

```

Here we use the Lua function `print()` provided by the `ngx_http_lua` module to generate NGINX error log messages at the `notice` level. This test case tests the number of the log messages containing the string `it is matched!`. It is important to note that only the *matched* part of the log file lines are collected in the final result instead of the whole log lines. This simplifies the comparison a lot since NGINX error log messages can contain varying details like timestamps and connection numbers.

A more useful form of this test is to specify a Perl regex pattern in the `grep_error_log` section. Consider the following example.

```

=== TEST 1: simple grep test for error logs
--- config
    location = /t {
        content_by_lua_block {
            print("test: before sleeping...")
            ngx.sleep(0.001) -- sleeping for 1ms
            print("test: after sleeping...")
        }
    }
--- request
GET /t
--- grep_error_log eval: qr/test: .*?\.\.\./
--- grep_error_log_out
test: before sleeping...
test: after sleeping...

```

We specify a Perl regex pattern, `test: .*?\.\.\./`, here to filter out all the error log messages starting with `test:` and ending with `...`. And naturally in this test we also require the relative order of these two messages, that is, `before sleeping` must appear *before* `after sleeping`. Otherwise, we shall see failure reports like below:

```
# Failed test 'TEST 1: simple grep test for error logs - grep_error'
# at ....lib/Test/Nginx/Socket.pm line 1048.
#     got: "test: after sleeping...\x{0a}test: before sleeping...\n"
# length: 49
# expected: "test: before sleeping...\x{0a}test: after sleeping...\n"
# length: 49
# strings begin to differ at char 7 (line 1 column 7)
```

As with the `response_body` section, we can also call the `no_long_string()` Perl function before `run_tests()` in the test file prologue, so as to disable the long string output mode and enable the `diff` mode. Then the test failure would look like this:

```
# Failed test 'TEST 1: simple grep test for error logs - grep_err'
# at ...lib/Test/Nginx/Socket.pm line 1044.
# @@ -1,2 +1,2 @@
# -test: before sleeping...
# test: after sleeping...
# +test: before sleeping...
```

Obviously, for this test case, the `diff` format looks better.

Extra Delay Before Log Checks

By default, `Test::Nginx::Socket` performs the NGINX error log checks not long after it receives the complete HTTP response for the test request. Sometimes, when the log messages are generated by the server after sending out the response, the error log checks may be carried out too early that the messages are not yet written into the log file. In this case, we can specify an extra delay via the `wait` data section for the test scaffold to wait for the error log messages. Here is an example:

```

=== TEST 1: wait for the timer
--- config
    location = /t {
        content_by_lua_block {
            local function f(premature)
                print("HERE!")
            end
            assert(ngx.timer.at(0.1, f))
        }
    }
--- request
GET /t
--- error_log
HERE!
--- no_error_log
[error]
--- wait: 0.12

```

Here we create a timer via the `ngx.timer.at` Lua function, which expires after 0.1 seconds. Due to the asynchronous nature of timers, the request handler does not wait for the timer to expire and immediately finishes processing the current request and sends out a response with an empty body. To check for the log message `HERE!` generated by the timer handler `f`, we have to specify an extra delay for the test scaffold to wait. The 0.12 seconds time is specified in this example but any values larger than 0.1 would suffice. Without the `wait` section, this test case would fail with the following output:

```
# Failed test 'TEST 1: wait for the timer - pattern "HERE!" matches
```

Obviously the test scaffold checks the error log too soon, even before the timer handler runs.

Section Review

`Test::Nginx::Socket` offers a rich set of data sections for specifying various different input data and expected output data, ranging from NGINX configuration file snippets, test requests, to expected responses and error log messages. We have already demonstrated the power of data driven testing and declarative test case crafting. We want to achieve multiple goals at the same time, that is, not only to make the tests self-contained and highly readable, but also to make the test report easy to interpret and analyze when some of the tests fail. Raw files automatically generated by the test scaffold, like

`t/servroot/conf/nginx.conf` and `t/servroot/logs/error.log`, should be checked frequently when manually debugging the test cases. The next section extends the discussion of this section with a focus on testing erroneous cases.

Testing Erroneous Cases

Most robust software invests heavily on error handling, and naturally test designers focus on corner cases and erroneous scenarios to maximize code coverage of the tests.

The previous section introduces data sections provided by `Test::Nginx::Socket` for examining messages in the NGINX error log file, which is a powerful tool to check for errors in the tests. Sometimes we want to test more extreme cases like server startup failures, malformed responses, bad requests, and various kinds of timeout errors.

Expected Server Startup Failures

Sometimes the NGINX server is expected to fail to start, like using an NGINX configuration directive in the wrong way or some hard prerequisites are not met in early initialization. If we want to test such cases, especially the error log messages generated for such failures, we could use the `must_die` data section in our test block to signal the test scaffold that the NGINX server is *expected* to die upon startup in this very block.

The following example tests the case of throwing a Lua exception in the context of `init_by_lua_block` of the `ngx_http_lua` module.

```
=== TEST 1: dying in init_by_lua_block
--- http_config
    init_by_lua_block {
        error("I am dying!")
    }
--- config
--- must_die
--- error_log
I am dying!
```

The Lua code in `init_by_lua_block` runs in the NGINX master process during the NGINX configuration file loading process. Throwing out a Lua exception there aborts the NGINX startup process immediately. The occurrence of the `must_die` section tells the test scaffold to treat NGINX server startup failures as a test pass while a successful startup as a test failure. The `error_log` section there ensures that the server fails in the expected way, that is, due to the "I am dying!" exception.

If we remove the `--- must_die` line from the test block above, then the test file won't even run to completion:

```
t/a.t .. nginx: [error] init_by_lua error: init_by_lua:2: I am dying
stack traceback:
  [C]: in function 'error'
  init_by_lua:2: in main chunk
Bailout called. Further testing stopped: TEST 1: dying in init_by
- Cannot start nginx using command
"nginx -p .../t/servroot/ -c .../t/servroot/conf/nginx.conf > /dev/
```

By default the test scaffold treats NGINX server startup failures as fatal errors in running the tests. The `must_die` section, however, turns such a failure into a normal test checkup.

Expected Malformed Responses

HTTP responses should always be well-formed, but unfortunately the real world is complicated and there indeed exists cases where the responses can be malformed, like being truncated due to some unexpected causes. As a test designer, we always want to test such strange abnormal cases, among other things.

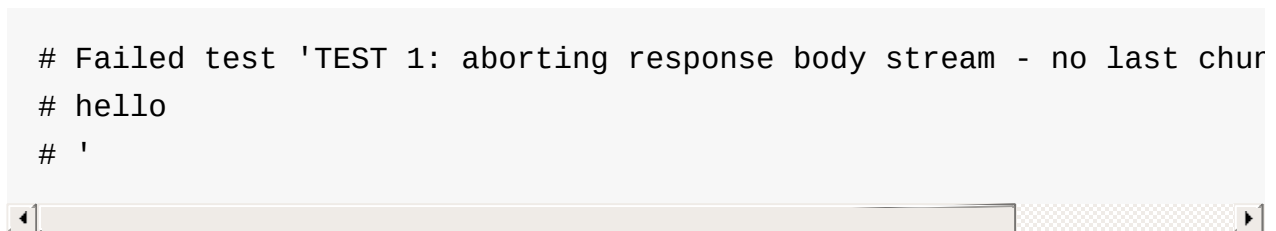
Naturally, `Test::Nginx::Socket` treats malformed responses from the NGINX server as an error since it always does sanity checks on the responses it receives from the test server by default. But for test cases where we expect a malformed or truncated response sent from the server, we should explicitly tell the test scaffold to disable the response sanity check via the `ignore_response` data section.

Consider the following example that closes the downstream connection immediately after sending out the first part of the response body.

```
=== TEST 1: aborting response body stream
--- config
    location = /t {
        content_by_lua_block {
            ngx.print("hello")
            ngx.flush(true)
            ngx.exit(444)
        }
    }
--- request
    GET /t
--- ignore_response
--- no_error_log
[error]
```

The `ngx.flush(true)` call in the `content_by_lua_block` handler is to ensure that any response body data buffered by NGINX is indeed flushed out to the system socket send buffers, which also usually means flushing the output data to the client side for local sockets. Also, the `ngx.exit(444)` call is used to immediately close the current downstream connection so it just interrupts the response body stream in the HTTP 1.1 chunked encoding. The important part is the `--- ignore_response` line which tells the test scaffold not to complain about the interrupted response data stream. If the test block above goes without this line, we will see the following test failure while running `prove` :

```
# Failed test 'TEST 1: aborting response body stream - no last chunk'
# hello
# '
```



Obviously, the test scaffold complains about the lack of the "last chunk" used to indicate the end of the chunked encoded data stream. Because the server aborts the connection in the middle of response body data sending, there is no chance for the server to properly send well-formed response bodies in the chunked encoding.

Testing Timeout Errors

Timeout errors are one of the most common network issues in the real world. Timeout might happen due to many reasons, like packet dropping on the wire or on the other end, connectivity problems, and other expensive operations blocking the event loop. Most of applications want to ensure that they have a timeout protection that prevents them from waiting for too long.

Testing and emulating timeout errors are often tricky in a self-contained unit test framework since most of the network traffic initiated by the test cases are local only, that is, going through the local "loopback" device that has perfect latency and throughput. We will examine some of the tricks that can be used to reliably emulate various different kinds of timeout errors in the test suite.

Connecting Timeouts

Connecting timeouts in the context of the TCP protocol are easiest to emulate. Just point the connecting target to a remote address that always drops any incoming (`SYN`) packets via a firewall rule or something similar. We provide such a "black-hole service" at the port 12345 of the `agentzh.org` host. You can make use of it if your test running environment allows public network access. Consider the following test case.

```

=== TEST 1: connect timeout
--- config
    resolver 8.8.8.8;
    resolver_timeout 1s;

    location = /t {
        content_by_lua_block {
            local sock = ngx.socket.tcp()
            sock:settimeout(100) -- ms
            local ok, err = sock:connect("agentzh.org", 12345)
            if not ok then
                ngx.log(ngx.ERR, "failed to connect: ", err)
                return ngx.exit(500)
            end
            ngx.say("ok")
        }
    }
--- request
GET /t
--- response_body_like: 500 Internal Server Error
--- error_code: 500
--- error_log
failed to connect: timeout

```

We have to configure the `resolver` directive here because we need to resolve the domain name `agentzh.org` at request time (in Lua). We check the NGINX error log via the `error_log` section for the error string returned by the `cosocket` object's `connect()` method.

It is important to use a relatively small timeout threshold in the test cases so that we do not have to wait for too long to complete the test run. Tests are meant to be run very often. The more frequently we run the tests, the more value we may gain from automating the tests.

It is worth mentioning that the test scaffold's HTTP client does have a timeout threshold as well, which is 3 seconds by default. If your test request takes more than 3 seconds, you get an error message in the test report:

```
ERROR: client socket timed out - TEST 1: connect timeout
```

This message is what we would get if we commented out the `settimeout` call and relies on the default 60 second timeout threshold in `cosockets`.

We could change this default timeout threshold used by the test scaffold client by setting a value to the `timeout` data section, as in

```
--- timeout: 10
```


Now we have 10 seconds of timeout protection instead of 3.

Reading Timeouts

Emulating reading timeouts is also easy. Just try reading from a wire where the other end never writes anything but still keeps the connection alive. Consider the following example:

```
=== TEST 1: read timeout
--- main_config
    stream {
        server {
            listen 5678;
            content_by_lua_block {
                ngx.sleep(10) -- 10 sec
            }
        }
    }
--- config
lua_socket_log_errors off;
location = /t {
    content_by_lua_block {
        local sock = ngx.socket.tcp()
        sock:settimeout(100) -- ms
        assert(sock:connect("127.0.0.1", 5678))
        ngx.say("connected.")
        local data, err = sock:receive() -- try to read a line
        if not data then
            ngx.say("failed to receive: ", err)
        else
            ngx.say("received: ", data)
        end
    }
}
--- request
GET /t
--- response_body
connected.
failed to receive: timeout
--- no_error_log
[error]
```

Here we use the `main_config` data section to define a TCP server of our own, listening at the port of 5678 on the local host. This is a mocked-up server that can establish new TCP connections but never write out anything and just sleep for 10 second before closing the session. Note that we are using the `ngx_stream_lua` module in the `stream {}` configuration block. In our `location = /t`, which is the main target of this test case, connects to our mock

server and tries to read a line from the wire. Apparently the 100ms timeout threshold on the client side is reached first and we can successfully exercise the error handling code for the reading timeout error.

Sending Timeouts

Triggering sending timeouts is much harder than connecting and reading timeouts. This is due to the asynchronous nature of writing.

For performance reasons, there exists at least two layers of buffers for writes:

1. the userland send buffers inside the NGINX core, and
2. the socket send buffers in the operating system kernel's TCP/IP stack implementation

To make the situation even worse, there also at least exists a system-level receive buffer layer on the other end of the connection.

To make a send timeout error happen, the most naive way is to fill out all these buffers along the data sending chain while ensuring that the other end never actually reads anything on the application level. Thus, buffering makes a sending timeout particularly hard to reproduce and emulate in a typical testing and development environment with a small amount of (test) payload.

Fortunately there is a userland trick that can intercept the libc wrappers for the actual system calls for socket I/O and do funny things that could otherwise be very difficult to achieve. Our [mockeagain](#) library implements such a trick and supports emulating timeout errors at user-specified precise positions in the output data stream.

The following example triggers a sending timeout right after sending out the "hello, world" string as the response body.

```

=== TEST 1: send timeout
--- config
    send_timeout 100ms;
    postpone_output 1;

    location = /t {
        content_by_lua_block {
            ngx.say("hi bob!")
            local ok, err = ngx.flush(true)
            if not ok then
                ngx.log(ngx.ERR, "flush #1 failed: ", err)
                return
            end

            ngx.say("hello, world!")
            local ok, err = ngx.flush(true)
            if not ok then
                ngx.log(ngx.ERR, "flush #2 failed: ", err)
                return
            end
        }
    }
--- request
GET /t
--- ignore_response
--- error_log
flush #2 failed: timeout
--- no_error_log
flush #1 failed

```

Note the `send_timeout` directive that is used to configure the sending timeout for NGINX downstream writing operations. Here we use a small threshold, `100ms`, to ensure our test case runs fast and never hits the default 3 seconds timeout threshold of the test scaffold client. The `postpone_output 1` directive effectively turns off the "postpone output buffer" of NGINX, which may hold our output data before even reaching the libc system call wrappers. Finally, the `ngx.flush()` call in Lua ensures that *no* buffers along the NGINX output filter chain holds our data without sending downward.

Before running this test case, we have to set the following system environment variables (in the bash syntax):

```

export LD_PRELOAD="mockeagain.so"
export MOCKEAGAIN="w"
export MOCKEAGAIN_WRITE_TIMEOUT_PATTERN='hello, world'
export TEST_NGINX_EVENT_TYPE='poll'

```

Let's go through them one by one:

1. The `LD_PRELOAD="mockeagain.so"` assignment pre-loads the `mockeagain` library into the running processes, including the NGINX server process started by the test scaffold, of course. You may also need to set the `LD_LIBRARY_PATH` environment to include the directory path of the `mockeagain.so` file if the file is not in the default system library search paths.
2. The `MOCKEAGAIN="w"` assignment enables the `mockeagain` library to intercept and do funny things about the writing operations on nonblocking sockets.
3. The `MOCKEAGAIN_WRITE_TIMEOUT_PATTERN='hello, world'` assignment makes `mockeagain` refuse to send more data after seeing the specified string pattern, `hello, world`, in the output data stream.
4. The `TEST_NGINX_EVENT_TYPE='poll'` setting makes NGINX server uses the `poll` event API instead of the system default (being `epoll` on Linux, for example). This is because `mockeagain` only supports `poll` events for now. Behind the scene, this environment just makes the test scaffold generate the following `nginx.conf` snippet.

```
events {  
    use poll;  
}
```

You need to ensure, however, that your NGINX or OpenResty build has the `poll` support compiled in. Basically, the build should have the `./configure` option `--with-poll_module`.

We have plans to add `epoll` edge-triggering support to `mockeagain` in the future. Hopefully by that time we do not have to use `poll` at least on Linux.

Now you should get the test block above passed!

Ideally, we could set these environments directly inside the test file because this test case will never pass without these environments anyway. We could add the following Perl code snippet to the very beginning of the test file prologue (yes, even before the `use` statement):

```
BEGIN {  
    $ENV{LD_PRELOAD} = "mockeagain.so";  
    $ENV{MOCKEAGAIN} = "w";  
    $ENV{MOCKEAGAIN_WRITE_TIMEOUT_PATTERN} = 'hello, world';  
    $ENV{TEST_NGINX_EVENT_TYPE} = 'poll';  
}
```

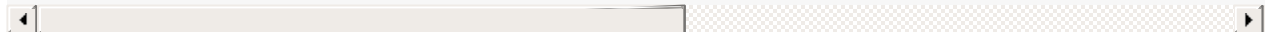
The `BEGIN {}` block is required here because it runs before Perl loads any modules, especially `Test::Nginx::Socket`, in which we want these environments to take affect.

It is a bad idea, however, to hard-code the path of the `mockeagain.so` file in the test file itself since different test runners might put `mockeagain` in different places in the file system. Better let the test runner configure the `LD_LIBRARY_PATH` environment containing the actual library path from outside.

Mockeagain Troubleshooting

If you are seeing the following error while running the test case above,

```
ERROR: ld.so: object 'mockeagain.so' from LD_PRELOAD cannot be prel
```

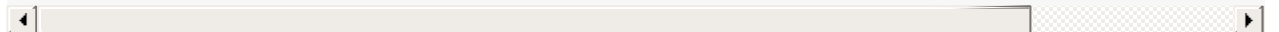


then you should check whether you have added the directory path of your `mockeagain.so` library to the `LD_LIBRARY_PATH` environment. On my system, for example, I have

```
export LD_LIBRARY_PATH=$HOME/git/mockeagain:$LD_LIBRARY_PATH
```

If you are seeing an error similar to the following,

```
nginx: [emerg] invalid event type "poll" in ../t/servroot/conf/ngi
```



then your NGINX or OpenResty build does not have the poll module compiled in. And you should rebuild your NGINX or OpenResty by passing the `--with-poll_module` option to the `./configure` command line.

We will revisit the `mockeagain` library in the `Test Modes` section soon.

Mocking Bad Backend Responses

Earlier in this section we have already seen examples that uses the `ngx_stream_lua` module to mock a backend TCP server that accepts new incoming connections but never writes anything back. We could of course do fancier things in such a mocked server like emulating a buggy or malicious backend server that returns bad response data.

For example, while testing a Memcached client, it would be pretty hard to emulate erroneous error responses or ill-formed responses with a real Memcached server. Now it is trivial with mocking:

```

=== TEST 1: get() results in an error response
--- main_config
    stream {
        server {
            listen 1921;
            content_by_lua_block {
                ngx.print("SERVER_ERROR\r\n")
            }
        }
    }
--- config
    location /t {
        content_by_lua_block {
            local memcached = require "resty.memcached"
            local memc = memcached:new()

            assert(memc:connect("127.0.0.1", 1921))

            local res, flags, err = memc:get("dog")
            if not res then
                ngx.say("failed to get: ", err)
                return
            end

            ngx.say("get: ", res)
            memc:close()
        }
    }
--- request
GET /t
--- response_body
failed to get: SERVER_ERROR
--- no_error_log
[error]

```

Our mocked-up Memcached server can behave in any way that we like. Hooray!

Note

`Test::Nginx::Socket` provides the data sections `tcp_listen`, `tcp_query`, `tcp_reply`, and etc to enable the builtin mocked TCP server of the test scaffold. You can use this facility when you do not want to depend on the `ngx_stream_lua` module or the NGINX stream subsystem for your test suite. Indeed, we were solely relying on the builtin TCP server of `Test::Nginx::Socket` before the `ngx_stream_lua` module was born. Similarly, `Test::Nginx::Socket` offers a builtin UDP server via the data sections `udp_listen`, `udp_query`, `udp_reply`, and etc. You can refer to the [official documentation](#) of `Test::Nginx::Socket` for more details.

Emulating Bad Clients

The `Test::Nginx::Socket` test framework provides special data sections to help emulating ill-behaved HTTP clients.

Crafting Bad Requests

The `raw_request` data section can be used to specify whatever data for the test request. It is often used with the `eval` section filter so that we can easily encode special characters like `\r`. Let's look at the following example.

```
=== TEST 1: missing the Host request header
--- config
    location = /t {
        return 200;
    }
--- raw_request eval
"GET /t HTTP/1.1\r
Connection: close\r
\r
"
--- response_body_like: 400 Bad Request
--- error_code: 400
```

So we easily construct a malformed request that does not have a `Host` header, which results in a 400 response from the NGINX server, as expected.

The `request` data section we have been using so far, on the other hand, always ensures that a well-formed HTTP request is sent to the test server.

Emulating Client Aborts

Client aborts are a very intriguing phenomenon in the web world. Sometimes we want the server to continue processing even after the client aborts the connection; on other occasions we just want to abort the whole request handler immediately in such cases. Either way, we need robust way to emulate client aborts in our unit test cases.

We have already discussed the `timeout` data section that can be used to adjust the default timeout protection threshold used by the test scaffold client. We could also use it to abort the connection prematurely. A small timeout threshold is often desired for this purpose. To suppress the test scaffold from printing out an error on client timeout, we can specify the `abort` data section to signal the test scaffold. Let's put these together in a simple test case.

```

=== TEST 1: abort processing in the Lua callback on client aborts
--- config
    location = /t {
        lua_check_client_abort on;

        content_by_lua_block {
            local ok, err = ngx.on_abort(function ()
                ngx.log(ngx.NOTICE, "on abort handler called!")
                ngx.exit(444)
            end)

            if not ok then
                error("cannot set on_abort: " .. err)
            end

            ngx.sleep(0.7) -- sec
            ngx.log(ngx.NOTICE, "main handler done")
        }
    }
--- request
    GET /t
--- timeout: 0.2
--- abort
--- ignore_response
--- no_error_log
[error]
main handler done
--- error_log
client prematurely closed connection
on abort handler called!

```

In this example, we make the test scaffold client abort the connection after 0.2 seconds via the `timeout` section. Also we prevent the test scaffold from printing out the client timeout error by specifying the `abort` section. Finally, in the Lua application code, we check for client abort events by turning on the `lua_check_client_abort` directive and aborts the server processing by calling `ngx.exit(444)` in our Lua callback function registered by the `ngx.on_abort` API.

Test Modes

One unique feature of `Test::Nginx` is that it allows running the same test suite in wildly different ways, or test modes, by just configuring some system environment variables. Different test modes have different focuses and may find different categories of bugs or performance issues in the applications being tested. The data driven nature of the test framework makes it easy to add new test modes without changing the user test files at all. And it is also possible to combine different test modes to form new (hybrid) test modes. The capability of running the same test suite in many different ways helps squeezing more value out of the tests we already have.

This section will iterate through various different test modes supported by `Test::Nginx::Socket` and their corresponding system environment variables used to enable or control them.

Benchmark Mode

HUP Reload Mode

Valgrind Mode

Naive Memory Leak Check Mode

Mockeagain Mode

Manual Debugging Mode

SystemTap Mode