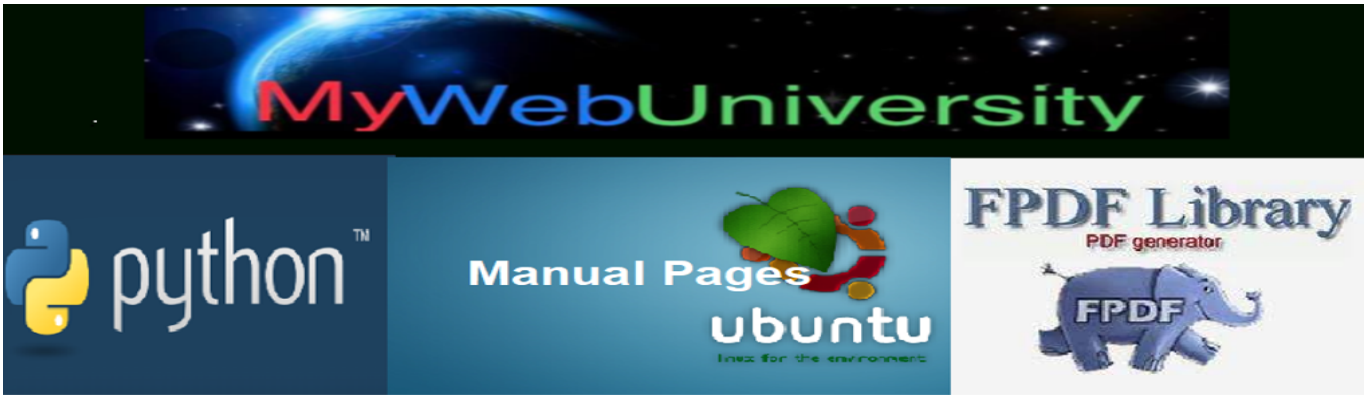

Rocky Enterprise Linux 9.2 Manual Pages on command 'Test2::API.3perl'

$ man Test2::API.3perl

Test2::API(3perl) Perl Programmers Reference Guide Test2::API(3perl)

NAME

Test2::API - Primary interface for writing Test2 based testing tools.

INTERNALS NOTE

The internals of this package are subject to change at any time! The public methods provided will not change in backwards-incompatible ways (once there is a stable release), but the underlying implementation details might. Do not break encapsulation here! Currently the implementation is to create a single instance of the Test2::API::Instance Object. All class methods defer to the single instance. There is no public access to the singleton, and that is intentional. The class methods provided by this package provide the only functionality publicly exposed.

This is done primarily to avoid the problems Test::Builder had by exposing its singleton. We do not want anyone to replace this singleton, rebless it, or directly muck with its internals. If you need to do something and cannot because of the restrictions placed here, then please report it as an issue. If possible, we will create a way for you to implement your functionality without exposing things that should not be exposed.

DESCRIPTION

This package exports all the functions necessary to write and/or verify testing tools. Using these building blocks you can begin writing test tools very quickly. You are also provided with tools that help you to test the tools you write.

SYNOPSIS

WRITING A TOOL

The "context()" method is your primary interface into the Test2 framework.

```
package My::Ok;
use Test2::API qw/context/;
our @EXPORT = qw/my_ok/;
use base 'Exporter';
# Just like ok() from Test::More
sub my_ok($;$) {
    my ($bool, $name) = @_;
    my $ctx = context(); # Get a context
    $ctx->ok($bool, $name);
    $ctx->release; # Release the context
    return $bool;
}
```

See Test2::API::Context for a list of methods available on the context object.

TESTING YOUR TOOLS

The "intercept { ... }" tool lets you temporarily intercept all events generated by the test system:

```
use Test2::API qw/intercept/;
use My::Ok qw/my_ok/;
my $events = intercept {
    # These events are not displayed
    my_ok(1, "pass");
    my_ok(0, "fail");
};
```

As of version 1.302178 this now returns an arrayref that is also an instance of Test2::API::InterceptResult. See the Test2::API::InterceptResult documentation for details on how to best use it.

OTHER API FUNCTIONS

```
use Test2::API qw{
    test2_init_done
    test2_stack
    test2_set_is_end
    test2_get_is_end
    test2_ipc
```

```
        test2_formatter_set
        test2_formatter
        test2_is_testing_done
    };
    my $init  = test2_init_done();
    my $stack = test2_stack();
    my $ipc   = test2_ipc();
    test2_formatter_set($FORMATTER)
    my $formatter = test2_formatter();
    ... And others ...
```

MAIN API EXPORTS

All exports are optional. You must specify subs to import.

```
    use Test2::API qw/context intercept run_subtest/;
```

This is the list of exports that are most commonly needed. If you are simply writing a tool, then this is probably all you need. If you need something and you cannot find it here, then you can also look at "OTHER API EXPORTS".

These exports lack the 'test2_' prefix because of how important/common they are. Exports in the "OTHER API EXPORTS" section have the 'test2_' prefix to ensure they stand out.

context(...)

Usage:

```
$ctx = context()
$ctx = context(%params)
```

The "context()" function will always return the current context. If there is already a context active, it will be returned. If there is not an active context, one will be generated. When a context is generated it will default to using the file and line number where the currently running sub was called from.

Please see "CRITICAL DETAILS" in Test2::API::Context for important rules about what you can and cannot do with a context once it is obtained.

Note This function will throw an exception if you ignore the context object it returns.

Note On perls 5.14+ a depth check is used to insure there are no context leaks. This cannot be safely done on older perls due to <https://rt.perl.org/Public/Bug/Display.html?id=127774> You can forcefully enable it either by setting "$ENV{T2_CHECK_DEPTH} = 1" or "$Test2::API::DO_DEPTH_CHECK = 1" BEFORE

loading Test2::API.

OPTIONAL PARAMETERS

All parameters to "context" are optional.

level => $int

If you must obtain a context in a sub deeper than your entry point you can use this to tell it how many EXTRA stack frames to look back. If this option is not provided the default of 0 is used.

```
sub third_party_tool {
    my $sub = shift;
    ... # Does not obtain a context
    $sub->();
    ...
}
third_party_tool(sub {
    my $ctx = context(level => 1);
    ...
    $ctx->release;
});
```

wrapped => $int

Use this if you need to write your own tool that wraps a call to "context()" with the intent that it should return a context object.

```
sub my_context {
    my %params = ( wrapped => 0, @_ );
    $params{wrapped}++;
    my $ctx = context(%params);
    ...
    return $ctx;
}
sub my_tool {
    my $ctx = my_context();
    ...
    $ctx->release;
}
```

If you do not do this, then tools you call that also check for a context will notice that the context they grabbed was created at the same stack depth, which will trigger protective measures that warn you and destroy the existing context.

stack => $stack

Normally "context()" looks at the global hub stack. If you are maintaining your own Test2::API::Stack instance you may pass it in to be used instead of the global one.

hub => $hub

Use this parameter if you want to obtain the context for a specific hub instead of whatever one happens to be at the top of the stack.

on_init => sub { ... }

This lets you provide a callback sub that will be called ONLY if your call to "context()" generated a new context. The callback WILL NOT be called if "context()" is returning an existing context. The only argument passed into the callback will be the context object itself.

```
sub foo {
    my $ctx = context(on_init => sub { 'will run' });
    my $inner = sub {
        # This callback is not run since we are getting the existing
        # context from our parent sub.
        my $ctx = context(on_init => sub { 'will NOT run' });
        $ctx->release;
    }
    $inner->();
    $ctx->release;
}
```

on_release => sub { ... }

This lets you provide a callback sub that will be called when the context instance is released. This callback will be added to the returned context even if an existing context is returned. If multiple calls to context add callbacks, then all will be called in reverse order when the context is finally released.

```
sub foo {
    my $ctx = context(on_release => sub { 'will run second' });
    my $inner = sub {
```

```
            my $ctx = context(on_release => sub { 'will run first' });
            # Neither callback runs on this release
            $ctx->release;
        }
        $inner->();
        # Both callbacks run here.
        $ctx->release;
    }
```

release($;$)

Usage:

```
release $ctx;
release $ctx, ...;
```

This is intended as a shortcut that lets you release your context and return a value in one statement. This function will get your context, and an optional return value. It will release your context, then return your value. Scalar context is always assumed.

```
sub tool {
    my $ctx = context();
    ...
    return release $ctx, 1;
}
```

This tool is most useful when you want to return the value you get from calling a function that needs to see the current context:

```
my $ctx = context();
my $out = some_tool(...);
$ctx->release;
return $out;
```

We can combine the last 3 lines of the above like so:

```
my $ctx = context();
release $ctx, some_tool(...);
```

context_do(&;@)

Usage:

```
sub my_tool {
    context_do {
```

```
            my $ctx = shift;
            my (@args) = @_;
            $ctx->ok(1, "pass");
            ...
            # No need to call $ctx->release, done for you on scope exit.
        } @_;
    }
```

Using this inside your test tool takes care of a lot of boilerplate for you. It will ensure a context is acquired. It will capture and rethrow any exception. It will insure the context is released when you are done. It preserves the subroutine call context (array, scalar, void).

This is the safest way to write a test tool. The only two downsides to this are a slight performance decrease, and some extra indentation in your source. If the indentation is a problem for you then you can take a peek at the next section.

no_context(&;$)

Usage:

```
no_context { ... };
no_context { ... } $hid;
        sub my_tool(&) {
            my $code = shift;
            my $ctx = context();
            ...
            no_context {
                # Things in here will not see our current context, they get a new
                # one.
                $code->();
            };
            ...
            $ctx->release;
        };
```

This tool will hide a context for the provided block of code. This means any tools run inside the block will get a completely new context if they acquire one. The new context will be inherited by tools nested below the one that acquired it.

This will normally hide the current context for the top hub. If you need to hide the context for a different hub you can pass in the optional $hid parameter.

intercept(&)

Usage:

```
my $events = intercept {
    ok(1, "pass");
    ok(0, "fail");
    ...
};
```

This function takes a codeblock as its only argument, and it has a prototype. It will execute the codeblock, intercepting any generated events in the process. It will return an array reference with all the generated event objects. All events should be subclasses of Test2::Event.

As of version 1.302178 the events array that is returned is blssed as an Test2::API::InterceptResult instance. Test2::API::InterceptResult Provides a helpful interface for filtering and/or inspecting the events list overall, or individual events within the list.

This is intended to help you test your test code. This is not intended for people simply writing tests.

run_subtest(...)

Usage:

```
run_subtest($NAME, \&CODE, $BUFFERED, @ARGS)
# or
run_subtest($NAME, \&CODE, \%PARAMS, @ARGS)
```

This will run the provided codeblock with the args in @args. This codeblock will be run as a subtest. A subtest is an isolated test state that is condensed into a single Test2::Event::Subtest event, which contains all events generated inside the subtest.

ARGUMENTS:

$NAME

The name of the subtest.

\&CODE

The code to run inside the subtest.

$BUFFERED or \%PARAMS

If this is a simple scalar then it will be treated as a boolean for the 'buffered' setting. If this is a hash reference then it will be used as a parameters hash. The param hash will be used for hub construction (with the specified keys removed).

Keys that are removed and used by run_subtest:

'buffered' => $bool

Toggle buffered status.

'inherit_trace' => $bool

Normally the subtest hub is pushed and the sub is allowed to generate its own root context for the hub. When this setting is turned on a root context will be created for the hub that shares the same trace as the current context.

Set this to true if your tool is producing subtests without user-specified subs.

'no_fork' => $bool

Defaults to off. Normally forking inside a subtest will actually fork the subtest, resulting in 2 final subtest events. This parameter will turn off that behavior, only the original process/thread will return a final subtest event.

@ARGS

Any extra arguments you want passed into the subtest code.

BUFFERED VS UNBUFFERED (OR STREAMED)

Normally all events inside and outside a subtest are sent to the formatter immediately by the hub. Sometimes it is desirable to hold off sending events within a subtest until the subtest is complete. This usually depends on the formatter being used.

Things not effected by this flag

In both cases events are generated and stored in an array. This array is eventually used to populate the "subevents" attribute on the Test2::Event::Subtest event that is generated at the end of the subtest. This flag has no effect on this part, it always happens.

At the end of the subtest, the final Test2::Event::Subtest event is sent to the formatter.

Things that are effected by this flag

The "buffered" attribute of the Test2::Event::Subtest event will be set to the value of this flag. This means any formatter, listener, etc which looks at the event will know if it was buffered.

Things that are formatter dependant

Events within a buffered subtest may or may not be sent to the formatter as they happen. If a formatter fails to specify then the default is to NOT SEND the events as they are generated, instead the formatter can pull them from the "subevents" attribute.

A formatter can specify by implementing the "hide_buffered()" method. If this method returns true then events generated inside a buffered subtest will not be sent independently of the final subtest event.

An example of how this is used is the Test2::Formatter::TAP formatter. For unbuffered subtests the events are rendered as they are generated. At the end of the subtest, the final subtest event is rendered, but the "subevents" attribute is ignored. For buffered subtests the opposite occurs, the events are NOT rendered as they are generated, instead the "subevents" attribute is used to render them all at once. This is useful when running subtests tests in parallel, since without it the output from subtests would be interleaved together.

OTHER API EXPORTS

Exports in this section are not commonly needed. These all have the 'test2_' prefix to help ensure they stand out. You should look at the "MAIN API EXPORTS" section before looking here. This section is one where "Great power comes with great responsibility". It is possible to break things badly if you are not careful with these.

All exports are optional. You need to list which ones you want at import time:

```
use Test2::API qw/test2_init_done .../;
```

STATUS AND INITIALIZATION STATE

These provide access to internal state and object instances.

$bool = test2_init_done()

This will return true if the stack and IPC instances have already been initialized. It will return false if they have not. Init happens as late as possible. It happens as soon as a tool requests the IPC instance, the formatter, or the stack.

$bool = test2_load_done()

This will simply return the boolean value of the loaded flag. If Test2 has finished loading this will be true, otherwise false. Loading is considered complete the first time a tool requests a context.

test2_set_is_end()

test2_set_is_end($bool)

This is used to toggle Test2's belief that the END phase has already started. With no arguments this will set it to true. With arguments it will set it to the first argument's value.

This is used to prevent the use of "caller()" in END blocks which can cause segfaults. This is only necessary in some persistent environments that may have multiple END phases.

$bool = test2_get_is_end()

Check if Test2 believes it is the END phase.

$stack = test2_stack()

This will return the global Test2::API::Stack instance. If this has not yet been initialized it will be initialized now.

$bool = test2_is_testing_done()

This will return true if testing is complete and no other events should be sent. This is useful in things like warning handlers where you might want to turn warnings into events, but need them to start acting like normal warnings when testing is done.

```
    $SIG{__WARN__} = sub {
        my ($warning) = @_;
        if (test2_is_testing_done()) {
            warn @_;
        }
        else {
            my $ctx = context();
            ...
            $ctx->release
        }
    }
```

test2_ipc_disable

Disable IPC.

$bool = test2_ipc_diabled

Check if IPC is disabled.

test2_ipc_wait_enable()

test2_ipc_wait_disable()

$bool = test2_ipc_wait_enabled()

These can be used to turn IPC waiting on and off, or check the current value of the flag.

Waiting is turned on by default. Waiting will cause the parent process/thread to wait until all child processes and threads are finished before exiting. You will almost never want to turn this off.

$bool = test2_no_wait()

test2_no_wait($bool)

DISCOURAGED: This is a confusing interface, it is better to use "test2_ipc_wait_enable()", "test2_ipc_wait_disable()" and "test2_ipc_wait_enabled()".

This can be used to get/set the no_wait status. Waiting is turned on by default. Waiting will cause the parent process/thread to wait until all child processes and threads are finished before exiting. You will almost never want to turn this off.

$fh = test2_stdout()

$fh = test2_stderr()

These functions return the filehandles that test output should be written to. They are primarily useful when writing a custom formatter and code that turns events into actual output (TAP, etc.). They will return a dupe of the original filehandles that formatted output can be sent to regardless of whatever state the currently running test may have left STDOUT and STDERR in.

test2_reset_io()

Re-dupe the internal filehandles returned by "test2_stdout()" and "test2_stderr()" from the current STDOUT and STDERR. You shouldn't need to do this except in very peculiar situations (for example, you're testing a new formatter and you need control over where the formatter is sending its output.)

BEHAVIOR HOOKS

These are hooks that allow you to add custom behavior to actions taken by Test2 and tools built on top of it.

test2_add_callback_exit(sub { ... })

This can be used to add a callback that is called after all testing is done. This is too late to add additional results, the main use of this callback is to set the exit code.

```
test2_add_callback_exit(
    sub {
```

```
        my ($context, $exit, \$new_exit) = @_;
        ...
    }
);
```

The $context passed in will be an instance of Test2::API::Context. The $exit argument will be the original exit code before anything modified it. $$new_exit is a reference to the new exit code. You may modify this to change the exit code. Please note that $$new_exit may already be different from $exit

test2_add_callback_post_load(sub { ... })

Add a callback that will be called when Test2 is finished loading. This means the callback will be run once, the first time a context is obtained. If Test2 has already finished loading then the callback will be run immediately.

test2_add_callback_testing_done(sub { ... })

This adds your coderef as a follow-up to the root hub after Test2 is finished loading. This is essentially a helper to do the following:

```
test2_add_callback_post_load(sub {
    my $stack = test2_stack();
    $stack->top; # Insure we have a hub
    my ($hub) = Test2::API::test2_stack->all;
    $hub->set_active(1);
    $hub->follow_up(sub { ... }); # <-- Your coderef here
});
```

test2_add_callback_context_acquire(sub { ... })

Add a callback that will be called every time someone tries to acquire a context. This will be called on EVERY call to "context()". It gets a single argument, a reference to the hash of parameters being used the construct the context. This is your chance to change the parameters by directly altering the hash.

```
test2_add_callback_context_acquire(sub {
    my $params = shift;
    $params->{level}++;
});
```

This is a very scary API function. Please do not use this unless you need to. This is here for Test::Builder and backwards compatibility. This has you directly manipulate

the hash instead of returning a new one for performance reasons.

test2_add_callback_context_init(sub { ... })

Add a callback that will be called every time a new context is created. The callback will receive the newly created context as its only argument.

test2_add_callback_context_release(sub { ... })

Add a callback that will be called every time a context is released. The callback will receive the released context as its only argument.

test2_add_callback_pre_subtest(sub { ... })

Add a callback that will be called every time a subtest is going to be run. The callback will receive the subtest name, coderef, and any arguments.

@list = test2_list_context_acquire_callbacks()

Return all the context acquire callback references.

@list = test2_list_context_init_callbacks()

Returns all the context init callback references.

@list = test2_list_context_release_callbacks()

Returns all the context release callback references.

@list = test2_list_exit_callbacks()

Returns all the exit callback references.

@list = test2_list_post_load_callbacks()

Returns all the post load callback references.

@list = test2_list_pre_subtest_callbacks()

Returns all the pre-subtest callback references.

test2_add_uuid_via(sub { ... })

$sub = test2_add_uuid_via()

This allows you to provide a UUID generator. If provided UUIDs will be attached to all events, hubs, and contexts. This is useful for storing, tracking, and linking these objects.

The sub you provide should always return a unique identifier. Most things will expect a proper UUID string, however nothing in Test2::API enforces this.

The sub will receive exactly 1 argument, the type of thing being tagged 'context', 'hub', or 'event'. In the future additional things may be tagged, in which case new strings will be passed in. These are purely informative, you can (and usually should) ignore them.

IPC AND CONCURRENCY

These let you access, or specify, the IPC system internals.

$bool = test2_has_ipc()

Check if IPC is enabled.

$ipc = test2_ipc()

This will return the global Test2::IPC::Driver instance. If this has not yet been initialized it will be initialized now.

test2_ipc_add_driver($DRIVER)

Add an IPC driver to the list. This will add the driver to the start of the list.

@drivers = test2_ipc_drivers()

Get the list of IPC drivers.

$bool = test2_ipc_polling()

Check if polling is enabled.

test2_ipc_enable_polling()

Turn on polling. This will cull events from other processes and threads every time a context is created.

test2_ipc_disable_polling()

Turn off IPC polling.

test2_ipc_enable_shm()

Legacy, this is currently a no-op that returns 0;

test2_ipc_set_pending($uniq_val)

Tell other processes and events that an event is pending. $uniq_val should be a unique value no other thread/process will generate.

Note: After calling this "test2_ipc_get_pending()" will return 1. This is intentional, and not avoidable.

$pending = test2_ipc_get_pending()

This returns -1 if there is no way to check (assume yes)

This returns 0 if there are (most likely) no pending events.

This returns 1 if there are (likely) pending events. Upon return it will reset, nothing else will be able to see that there were pending events.

$timeout = test2_ipc_get_timeout()

test2_ipc_set_timeout($timeout)

Get/Set the timeout value for the IPC system. This timeout is how long the IPC system

will wait for child processes and threads to finish before aborting.

The default value is 30 seconds.

MANAGING FORMATTERS

These let you access, or specify, the formatters that can/should be used.

$formatter = test2_formatter

This will return the global formatter class. This is not an instance. By default the formatter is set to Test2::Formatter::TAP.

You can override this default using the "T2_FORMATTER" environment variable.

Normally 'Test2::Formatter::' is prefixed to the value in the environment variable:

```
$ T2_FORMATTER='TAP' perl test.t     # Use the Test2::Formatter::TAP formatter
$ T2_FORMATTER='Foo' perl test.t     # Use the Test2::Formatter::Foo formatter
```

If you want to specify a full module name you use the '+' prefix:

```
$ T2_FORMATTER='+Foo::Bar' perl test.t     # Use the Foo::Bar formatter
```

test2_formatter_set($class_or_instance)

Set the global formatter class. This can only be set once. Note: This will override anything specified in the 'T2_FORMATTER' environment variable.

@formatters = test2_formatters()

Get a list of all loaded formatters.

test2_formatter_add($class_or_instance)

Add a formatter to the list. Last formatter added is used at initialization. If this is called after initialization a warning will be issued.

OTHER EXAMPLES

See the "/Examples/" directory included in this distribution.

SEE ALSO

Test2::API::Context - Detailed documentation of the context object.

Test2::IPC - The IPC system used for threading/fork support.

Test2::Formatter - Formatters such as TAP live here.

Test2::Event - Events live in this namespace.

Test2::Hub - All events eventually funnel through a hub. Custom hubs are how "intercept()" and "run_subtest()" are implemented.

MAGIC

This package has an END block. This END block is responsible for setting the exit code based on the test results. This end block also calls the callbacks that can be added to

this package.

SOURCE

The source code repository for Test2 can be found at

http://github.com/Test-More/test-more/.

MAINTAINERS

Chad Granum <exodist@cpan.org>

AUTHORS

Chad Granum <exodist@cpan.org>

COPYRIGHT