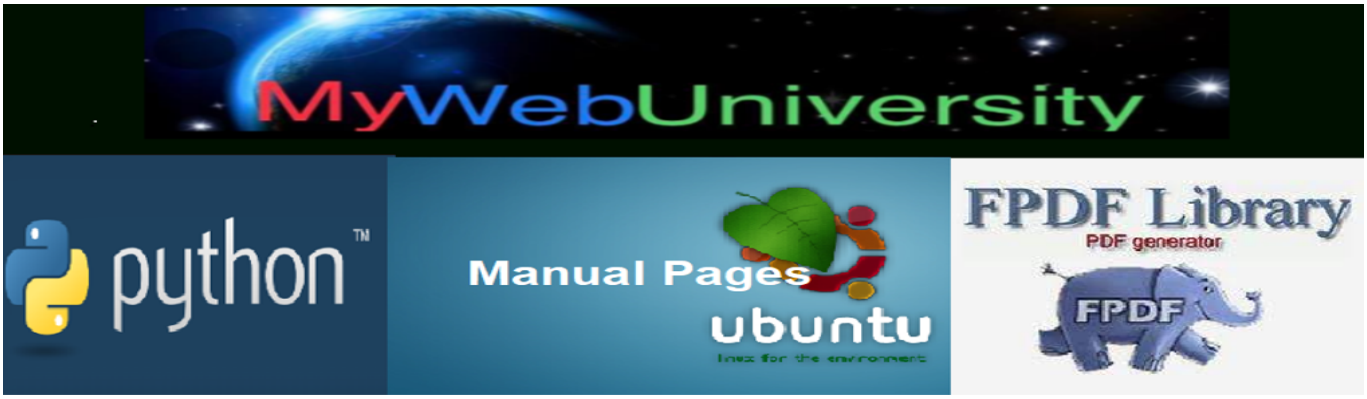

Rocky Enterprise Linux 9.2 Manual Pages on command 'Test2::API::Context.3perl'

$ man Test2::API::Context.3perl

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

NAME

Test2::API::Context - Object to represent a testing context.

DESCRIPTION

The context object is the primary interface for authors of testing tools written with Test2. The context object represents the context in which a test takes place (File and Line Number), and provides a quick way to generate events from that context. The context object also takes care of sending events to the correct Test2::Hub instance.

SYNOPSIS

In general you will not be creating contexts directly. To obtain a context you should always use "context()" which is exported by the Test2::API module.

```
use Test2::API qw/context/;
sub my_ok {
    my ($bool, $name) = @_;
    my $ctx = context();
    if ($bool) {
        $ctx->pass($name);
    }
    else {
        $ctx->fail($name);
    }
    $ctx->release; # You MUST do this!
    return $bool;
```

```
}
```

Context objects make it easy to wrap other tools that also use context. Once you grab a context, any tool you call before releasing your context will inherit it:

```
sub wrapper {
    my ($bool, $name) = @_;
    my $ctx = context();
    $ctx->diag("wrapping my_ok");
    my $out = my_ok($bool, $name);
    $ctx->release; # You MUST do this!
    return $out;
}
```

CRITICAL DETAILS

you MUST always use the context() sub from Test2::API

Creating your own context via "Test2::API::Context->new()" will almost never produce a desirable result. Use "context()" which is exported by Test2::API.

There are a handful of cases where a tool author may want to create a new context by hand, which is why the "new" method exists. Unless you really know what you are doing you should avoid this.

You MUST always release the context when done with it

Releasing the context tells the system you are done with it. This gives it a chance to run any necessary callbacks or cleanup tasks. If you forget to release the context it will try to detect the problem and warn you about it.

You MUST NOT pass context objects around

When you obtain a context object it is made specifically for your tool and any tools nested within. If you pass a context around you run the risk of polluting other tools with incorrect context information.

If you are certain that you want a different tool to use the same context you may pass it a snapshot. "$ctx->snapshot" will give you a shallow clone of the context that is safe to pass around or store.

You MUST NOT store or cache a context for later

As long as a context exists for a given hub, all tools that try to get a context will get the existing instance. If you try to store the context you will pollute other tools with incorrect context information.

If you are certain that you want to save the context for later, you can use a snapshot. "$ctx->snapshot" will give you a shallow clone of the context that is safe to pass around or store.

"context()" has some mechanisms to protect you if you do cause a context to persist beyond the scope in which it was obtained. In practice you should not rely on these protections, and they are fairly noisy with warnings.

You SHOULD obtain your context as soon as possible in a given tool

You never know what tools you call from within your own tool will need a context. Obtaining the context early ensures that nested tools can find the context you want them to find.

METHODS

$ctx->done_testing;

Note that testing is finished. If no plan has been set this will generate a Plan event.

$clone = $ctx->snapshot()

This will return a shallow clone of the context. The shallow clone is safe to store for later.

$ctx->release()

This will release the context. This runs cleanup tasks, and several important hooks. It will also restore $!, $?, and $@ to what they were when the context was created.

Note: If a context is acquired more than once an internal refcount is kept. "release()" decrements the ref count, none of the other actions of "release()" will occur unless the refcount hits 0. This means only the last call to "release()" will reset $?, $!, $@,and run the cleanup tasks.

$ctx->throw($message)

This will throw an exception reporting to the file and line number of the context. This will also release the context for you.

$ctx->alert($message)

This will issue a warning from the file and line number of the context.

$stack = $ctx->stack()

This will return the Test2::API::Stack instance the context used to find the current hub.

$hub = $ctx->hub()

This will return the Test2::Hub instance the context recognizes as the current one to which all events should be sent.

$dbg = $ctx->trace()

This will return the Test2::EventFacet::Trace instance used by the context.

$ctx->do_in_context(\&code, @args);

Sometimes you have a context that is not current, and you want things to use it as the current one. In these cases you can call "$ctx->do_in_context(sub { ... })". The codeblock will be run, and anything inside of it that looks for a context will find the one on which the method was called.

This DOES NOT affect context on other hubs, only the hub used by the context will be affected.

```
my $ctx = ...;
$ctx->do_in_context(sub {
    my $ctx = context(); # returns the $ctx the sub is called on
});
```

Note: The context will actually be cloned, the clone will be used instead of the original. This allows the thread id, process id, and error variables to be correct without modifying the original context.

$ctx->restore_error_vars()

This will set $!, $?, and $@ to what they were when the context was created. There is no localization or anything done here, calling this method will actually set these vars.

$! = $ctx->errno()

The (numeric) value of $! when the context was created.

$? = $ctx->child_error()

The value of $? when the context was created.

$@ = $ctx->eval_error()

The value of $@ when the context was created.

EVENT PRODUCTION METHODS

Which one do I use?

The "pass*" and "fail*" are optimal if they meet your situation, using one of them will always be the most optimal. That said they are optimal by eliminating many features.

Method such as "ok", and "note" are shortcuts for generating common 1-task events based on

the old API, however they are forward compatible, and easy to use. If these meet your needs then go ahead and use them, but please check back often for alternatives that may be added.

If you want to generate new style events, events that do many things at once, then you want the "*ev2*" methods. These let you directly specify which facets you wish to use.

$event = $ctx->pass()

$event = $ctx->pass($name)

This will send and return an Test2::Event::Pass event. You may optionally provide a $name for the assertion.

The Test2::Event::Pass is a specially crafted and optimized event, using this will help the performance of passing tests.

$true = $ctx->pass_and_release()

$true = $ctx->pass_and_release($name)

This is a combination of "pass()" and "release()". You can use this if you do not plan to do anything with the context after sending the event. This helps write more clear and compact code.

```
sub shorthand {
    my ($bool, $name) = @_;
    my $ctx = context();
    return $ctx->pass_and_release($name) if $bool;
    ... Handle a failure ...
}
sub longform {
    my ($bool, $name) = @_;
    my $ctx = context();
    if ($bool) {
        $ctx->pass($name);
        $ctx->release;
        return 1;
    }
    ... Handle a failure ...
}
```

my $event = $ctx->fail()

my $event = $ctx->fail($name)

my $event = $ctx->fail($name, @diagnostics)

This lets you send an Test2::Event::Fail event. You may optionally provide a $name and @diagnostics messages.

Diagnostics messages can be simple strings, data structures, or instances of Test2::EventFacet::Info::Table (which are converted inline into the Test2::EventFacet::Info structure).

my $false = $ctx->fail_and_release()

my $false = $ctx->fail_and_release($name)

my $false = $ctx->fail_and_release($name, @diagnostics)

This is a combination of "fail()" and "release()". This can be used to write clearer and shorter code.

```
sub shorthand {
    my ($bool, $name) = @_;
    my $ctx = context();
    return $ctx->fail_and_release($name) unless $bool;
    ... Handle a success ...
}
sub longform {
    my ($bool, $name) = @_;
    my $ctx = context();
    unless ($bool) {
        $ctx->pass($name);
        $ctx->release;
        return 1;
    }
    ... Handle a success ...
}
```

$event = $ctx->ok($bool, $name)

$event = $ctx->ok($bool, $name, \@on_fail)

NOTE: Use of this method is discouraged in favor of "pass()" and "fail()" which produce Test2::Event::Pass and Test2::Event::Fail events. These newer event types are faster and less crufty.

This will create an Test2::Event::Ok object for you. If $bool is false then an Test2::Event::Diag event will be sent as well with details about the failure. If you do not want automatic diagnostics you should use the "send_event()" method directly.

The third argument "\@on_fail") is an optional set of diagnostics to be sent in the event of a test failure. Unlike with "fail()" these diagnostics must be plain strings, data structures are not supported.

$event = $ctx->note($message)

Send an Test2::Event::Note. This event prints a message to STDOUT.

$event = $ctx->diag($message)

Send an Test2::Event::Diag. This event prints a message to STDERR.

$event = $ctx->plan($max)

$event = $ctx->plan(0, 'SKIP', $reason)

This can be used to send an Test2::Event::Plan event. This event usually takes either a number of tests you expect to run. Optionally you can set the expected count to 0 and give the 'SKIP' directive with a reason to cause all tests to be skipped.

$event = $ctx->skip($name, $reason);

Send an Test2::Event::Skip event.

$event = $ctx->bail($reason)

This sends an Test2::Event::Bail event. This event will completely terminate all testing.

$event = $ctx->send_ev2(%facets)

This lets you build and send a V2 event directly from facets. The event is returned after it is sent.

This example sends a single assertion, a note (comment for stdout in Test::Builder talk) and sets the plan to 1.

```
my $event = $ctx->send_event(
    plan   => {count => 1},
    assert => {pass  => 1, details => "A passing assert"},
    info => [{tag => 'NOTE', details => "This is a note"}],
);
```

$event = $ctx->build_e2(%facets)

This is the same as "send_ev2()", except it builds and returns the event without sending it.

$event = $ctx->send_ev2_and_release($Type, %parameters)

This is a combination of "send_ev2()" and "release()".

```
sub shorthand {
    my $ctx = context();
    return $ctx->send_ev2_and_release(assert => {pass => 1, details => 'foo'});
}
sub longform {
    my $ctx = context();
    my $event = $ctx->send_ev2(assert => {pass => 1, details => 'foo'});
    $ctx->release;
    return $event;
}
```

$event = $ctx->send_event($Type, %parameters)

It is better to use send_ev2() in new code.

This lets you build and send an event of any type. The $Type argument should be the event package name with "Test2::Event::" left off, or a fully qualified package name prefixed with a '+'. The event is returned after it is sent.

```
my $event = $ctx->send_event('Ok', ...);
```

or

```
my $event = $ctx->send_event('+Test2::Event::Ok', ...);
```

$event = $ctx->build_event($Type, %parameters)

It is better to use build_ev2() in new code.

This is the same as "send_event()", except it builds and returns the event without sending it.

$event = $ctx->send_event_and_release($Type, %parameters)

It is better to use send_ev2_and_release() in new code.

This is a combination of "send_event()" and "release()".

```
sub shorthand {
    my $ctx = context();
    return $ctx->send_event_and_release(Pass => { name => 'foo' });
}
sub longform {
    my $ctx = context();
```

```
        my $event = $ctx->send_event(Pass => { name => 'foo' });
        $ctx->release;
        return $event;
    }
```

HOOKS

There are 2 types of hooks, init hooks, and release hooks. As the names suggest, these hooks are triggered when contexts are created or released.

INIT HOOKS

These are called whenever a context is initialized. That means when a new instance is created. These hooks are NOT called every time something requests a context, just when a new one is created.

GLOBAL

This is how you add a global init callback. Global callbacks happen for every context for any hub or stack.

```
    Test2::API::test2_add_callback_context_init(sub {
        my $ctx = shift;
        ...
    });
```

PER HUB

This is how you add an init callback for all contexts created for a given hub. These callbacks will not run for other hubs.

```
    $hub->add_context_init(sub {
        my $ctx = shift;
        ...
    });
```

PER CONTEXT

This is how you specify an init hook that will only run if your call to "context()" generates a new context. The callback will be ignored if "context()" is returning an existing context.

```
    my $ctx = context(on_init => sub {
        my $ctx = shift;
        ...
    });
```

RELEASE HOOKS

These are called whenever a context is released. That means when the last reference to the instance is about to be destroyed. These hooks are NOT called every time "$ctx->release" is called.

GLOBAL

This is how you add a global release callback. Global callbacks happen for every context for any hub or stack.

```
Test2::API::test2_add_callback_context_release(sub {
    my $ctx = shift;
    ...
});
```

PER HUB

This is how you add a release callback for all contexts created for a given hub. These callbacks will not run for other hubs.

```
$hub->add_context_release(sub {
    my $ctx = shift;
    ...
});
```

PER CONTEXT

This is how you add release callbacks directly to a context. The callback will ALWAYS be added to the context that gets returned, it does not matter if a new one is generated, or if an existing one is returned.

```
my $ctx = context(on_release => sub {
    my $ctx = shift;
    ...
});
```

THIRD PARTY META-DATA

This object consumes Test2::Util::ExternalMeta which provides a consistent way for you to attach meta-data to instances of this class. This is useful for tools, plugins, and other extensions.

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>

Kent Fredric <kentnl@cpan.org>

COPYRIGHT