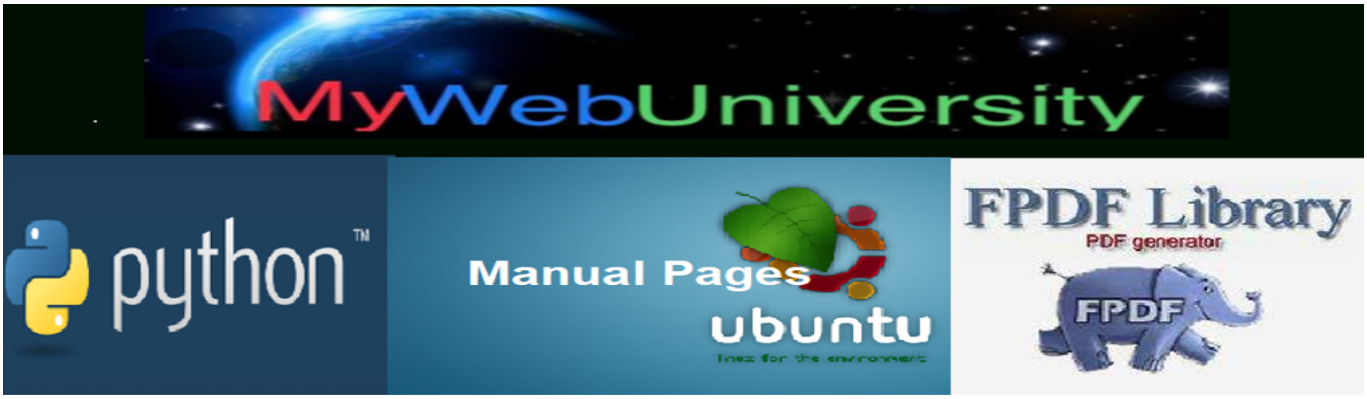

Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

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

$ man Test2::Event.3perl

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

NAME

Test2::Event - Base class for events

DESCRIPTION

Base class for all event objects that get passed through Test2.

SYNOPSIS

```
package Test2::Event::MyEvent;

use strict;

use warnings;

# This will make our class an event subclass (required)

use base 'Test2::Event';

# Add some accessors (optional)

# You are not obligated to use HashBase, you can use any object tool you

# want, or roll your own accessors.

use Test2::Util::HashBase qw/foo bar baz/;

# Use this if you want the legacy API to be written for you, for this to

# work you will need to implement a facet_data() method.

use Test2::Util::Facets2Legacy;

# Chance to initialize some defaults

sub init {

    my $self = shift;

    # no other args in @_

    $self->set_foo('xxx') unless defined $self->foo;
```

```
    ...
}
# This is the new way for events to convey data to the Test2 system
sub facet_data {
    my $self = shift;
    # Get common facets such as 'about', 'trace' 'amnesty', and 'meta'
    my $facet_data = $self->common_facet_data();
    # Are you making an assertion?
    $facet_data->{assert} = {pass => 1, details => 'my assertion'};
    ...
    return $facet_data;
}
1;
```

METHODS

GENERAL

$trace = $e->trace

Get a snapshot of the Test2::EventFacet::Trace as it was when this event was generated

$bool_or_undef = $e->related($e2)

Check if 2 events are related. In this case related means their traces share a signature meaning they were created with the same context (or at the very least by contexts which share an id, which is the same thing unless someone is doing something very bad).

This can be used to reliably link multiple events created by the same tool. For instance a failing test like "ok(0, "fail"" will generate 2 events, one being a Test2::Event::Ok, the other being a Test2::Event::Diag, both of these events are related having been created under the same context and by the same initial tool (though multiple tools may have been nested under the initial one).

This will return "undef" if the relationship cannot be checked, which happens if either event has an incomplete or missing trace. This will return 0 if the traces are complete, but do not match. 1 will be returned if there is a match.

$e->add_amnesty({tag => $TAG, details => $DETAILS});

This can be used to add amnesty to this event. Amnesty only effects failing assertions in most cases, but some formatters may display them for passing assertions, or even

non-assertions as well.

Amnesty will prevent a failed assertion from causing the overall test to fail. In other words it marks a failure as expected and allowed.

Note: This is how 'TODO' is implemented under the hood. TODO is essentially amnesty with the 'TODO' tag. The details are the reason for the TODO.

$uuid = $e->uuid

If UUID tagging is enabled (See Test::API) then any event that has made its way through a hub will be tagged with a UUID. A newly created event will not yet be tagged in most cases.

$class = $e->load_facet($name)

This method is used to load a facet by name (or key). It will attempt to load the facet class, if it succeeds it will return the class it loaded. If it fails it will return "undef". This caches the result at the class level so that future calls will be faster.

The $name variable should be the key used to access the facet in a facets hashref. For instance the assertion facet has the key 'assert', the information facet has the 'info' key, and the error facet has the key 'errors'. You may include or omit the 's' at the end of the name, the method is smart enough to try both the 's' and no-'s' forms, it will check what you provided first, and if that is not found it will add or strip the 's and try again.

@classes = $e->FACET_TYPES()

@classes = Test2::Event->FACET_TYPES()

This returns a list of all facets that have been loaded using the "load_facet()" method. This will not return any classes that have not been loaded, or have been loaded directly without a call to "load_facet()".

Note: The core facet types are automatically loaded and populated in this list.

NEW API

$hashref = $e->common_facet_data();

This can be used by subclasses to generate a starting facet data hashref. This will populate the hashref with the trace, meta, amnesty, and about facets. These facets are nearly always produced the same way for all events.

$hashref = $e->facet_data()

If you do not override this then the default implementation will attempt to generate

facets from the legacy API. This generation is limited only to what the legacy API can provide. It is recommended that you override this method and write out explicit facet data.

$hashref = $e->facets()

This takes the hashref from "facet_data()" and blesses each facet into the proper "Test2::EventFacet::*" subclass. If no class can be found for any given facet it will be passed along unchanged.

@errors = $e->validate_facet_data();

@errors = $e->validate_facet_data(%params);

@errors = $e->validate_facet_data(\%facets, %params);

@errors = Test2::Event->validate_facet_data(%params);

@errors = Test2::Event->validate_facet_data(\%facets, %params);

This method will validate facet data and return a list of errors. If no errors are found this will return an empty list.

This can be called as an object method with no arguments, in which case the "facet_data()" method will be called to get the facet data to be validated.

When used as an object method the "\%facet_data" argument may be omitted.

When used as a class method the "\%facet_data" argument is required.

Remaining arguments will be slurped into a %params hash.

Currently only 1 parameter is defined:

require_facet_class => $BOOL

When set to true (default is false) this will reject any facets where a facet class cannot be found. Normally facets without classes are assumed to be custom and are ignored.

WHAT ARE FACETS?

Facets are how events convey their purpose to the Test2 internals and formatters. An event without facets will have no intentional effect on the overall test state, and will not be displayed at all by most formatters, except perhaps to say that an event of an unknown type was seen.

Facets are produced by the "facet_data()" subroutine, which you should nearly-always override. "facet_data()" is expected to return a hashref where each key is the facet type, and the value is either a hashref with the data for that facet, or an array of hashrefs.

Some facets must be defined as single hashrefs, some must be defined as an array of

hashrefs, No facets allow both.

"facet_data()" MUST NOT bless the data it returns, the main hashref, and nested facet hashrefs MUST be bare, though items contained within each facet may be blessed. The data returned by this method should also be copies of the internal data in order to prevent accidental state modification.

"facets()" takes the data from "facet_data()" and blesses it into the "Test2::EventFacet::*" packages. This is rarely used however, the EventFacet packages are primarily for convenience and documentation. The EventFacet classes are not used at all internally, instead the raw data is used.

Here is a list of facet types by package. The packages are not used internally, but are where the documentation for each type is kept.

Note: Every single facet type has the 'details' field. This field is always intended for human consumption, and when provided, should explain the 'why' for the facet. All other fields are facet specific.

about => {...}

Test2::EventFacet::About

This contains information about the event itself such as the event package name. The "details" field for this facet is an overall summary of the event.

assert => {...}

Test2::EventFacet::Assert

This facet is used if an assertion was made. The "details" field of this facet is the description of the assertion.

control => {...}

Test2::EventFacet::Control

This facet is used to tell the Test2::Event::Hub about special actions the event causes. Things like halting all testing, terminating the current test, etc. In this facet the "details" field explains why any special action was taken.

Note: This is how bail-out is implemented.

meta => {...}

Test2::EventFacet::Meta

The meta facet contains all the meta-data attached to the event. In this case the "details" field has no special meaning, but may be present if something sets the 'details' meta-key on the event.

parent => {...}

Test2::EventFacet::Parent

This facet contains nested events and similar details for subtests. In this facet the "details" field will typically be the name of the subtest.

plan => {...}

Test2::EventFacet::Plan

This facet tells the system that a plan has been set. The "details" field of this is usually left empty, but when present explains why the plan is what it is, this is most useful if the plan is to skip-all.

trace => {...}

Test2::EventFacet::Trace

This facet contains information related to when and where the event was generated. This is how the test file and line number of a failure is known. This facet can also help you to tell if tests are related.

In this facet the "details" field overrides the "failed at test_file.t line 42." message provided on assertion failure.

amnesty => [{...}, ...]

Test2::EventFacet::Amnesty

The amnesty facet is a list instead of a single item, this is important as amnesty can come from multiple places at once.

For each instance of amnesty the "details" field explains why amnesty was granted.

Note: Outside of formatters amnesty only acts to forgive a failing assertion.

errors => [{...}, ...]

Test2::EventFacet::Error

The errors facet is a list instead of a single item, any number of errors can be listed. In this facet "details" describes the error, or may contain the raw error message itself (such as an exception). In perl exception may be blessed objects, as such the raw data for this facet may contain nested items which are blessed.

Not all errors are considered fatal, there is a "fail" field that must be set for an error to cause the test to fail.

Note: This facet is unique in that the field name is 'errors' while the package is 'Error'. This is because this is the only facet type that is both a list, and has a name where the plural is not the same as the singular. This may cause some confusion,

but I feel it will be less confusing than the alternative.

info => [{...}, ...]

Test2::EventFacet::Info

The 'info' facet is a list instead of a single item, any quantity of extra information can be attached to an event. Some information may be critical diagnostics, others may be simply commentary in nature, this is determined by the "debug" flag.

For this facet the "details" flag is the info itself. This info may be a string, or it may be a data structure to display. This is one of the few facet types that may contain blessed items.

LEGACY API

$bool = $e->causes_fail

Returns true if this event should result in a test failure. In general this should be false.

$bool = $e->increments_count

Should be true if this event should result in a test count increment.

$e->callback($hub)

If your event needs to have extra effects on the Test2::Hub you can override this method.

This is called BEFORE your event is passed to the formatter.

$num = $e->nested

If this event is nested inside of other events, this should be the depth of nesting. (This is mainly for subtests)

$bool = $e->global

Set this to true if your event is global, that is ALL threads and processes should see it no matter when or where it is generated. This is not a common thing to want, it is used by bail-out and skip_all to end testing.

$code = $e->terminate

This is called AFTER your event has been passed to the formatter. This should normally return undef, only change this if your event should cause the test to exit immediately.

If you want this event to cause the test to exit you should return the exit code here. Exit code of 0 means exit success, any other integer means exit with failure.

This is used by Test2::Event::Plan to exit 0 when the plan is 'skip_all'. This is also

used by Test2::Event:Bail to force the test to exit with a failure.

This is called after the event has been sent to the formatter in order to ensure the event is seen and understood.

$msg = $e->summary

This is intended to be a human readable summary of the event. This should ideally only be one line long, but you can use multiple lines if necessary. This is intended for human consumption. You do not need to make it easy for machines to understand.

The default is to simply return the event package name.

($count, $directive, $reason) = $e->sets_plan()

Check if this event sets the testing plan. It will return an empty list if it does not. If it does set the plan it will return a list of 1 to 3 items in order: Expected Test Count, Test Directive, Reason for directive.

$bool = $e->diagnostics

True if the event contains diagnostics info. This is useful because a non-verbose harness may choose to hide events that are not in this category. Some formatters may choose to send these to STDERR instead of STDOUT to ensure they are seen.

$bool = $e->no_display

False by default. This will return true on events that should not be displayed by formatters.

$id = $e->in_subtest

If the event is inside a subtest this should have the subtest ID.

$id = $e->subtest_id

If the event is a final subtest event, this should contain the subtest ID.

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>

COPYRIGHT