NAME

Fennec − A testers toolbox, and best friend

DESCRIPTION

Fennec ties together several testing related modules and enhances their functionality in ways you don't get loading them individually. Fennec makes testing easier, and more useful.

SYNOPSYS

There are 2 ways to use Fennec. You can use Fennec directly, or you can use the shiny sugar-coated interface provided by the add-on module Fennec::Declare.

VANILLA SYNTAX

If Devel::Declare and its awesome power of syntax specification scares you, you can always write your Fennec tests in the stone age like this... just don't miss any semicolons.

t/some_test.t:

```
package TEST::SomeTest;
use strict;
use warnings;

use Fennec(
    parallel  => 3,
    test_sort => 'random',
);

# This is optional, there is a default 'new' if you do not override it.
sub new { ... }

# Test blocks are called as methods on an instance of your test package.
tests group_1 => sub {
    my $self = shift;
    ok( 1, "1 is true" );
};

test group_2 => (
    todo => 'This is not ready yet',
    code => sub {
        my $self = shift;
        ok( 0, "Not ready" );
    }
);

# It is important to always end a Fennec test with this function call.
done_testing();
```

DECLARE SYNTAX

Note: In order to use this you **MUST** install Fennec::Declare which is a separate distribution on cpan. This module is separate because it uses the controversial Devel::Declare module.

t/some_test.t:

```
package TEST::SomeTest;
use strict;
use warnings;

use Fennec::Declare(
    parallel  => 3,
    test_sort => 'random',
);

# This is optional, there is a default 'new' if you do not override it.
sub new { ... }

# Test blocks are called as methods on an instance of your test package.
```

```
tests group_1 {
    # Note: $self is automatically shifted for you.
    ok( $self, "Got self automatically" );
};

test group_2 ( todo => 'This is not ready yet' ) {
    # Note: $self is automatically shifted for you.
    ok( 0, "Not ready" );
}

# It is important to always end a Fennec test with this function call.
done_testing;
```

FEATURES

PROVIDED DIRECTLY BY FENNEC

Forking just works

Forking in perl tests that use Test::Builder is perilous at best. Fennec initiates an Fennec::Collector class which sets up Test::Builder to funnel all test results to the main thread for rendering. A result of this is that forking just works.

Concurrency, test blocks can run in parallel

By default all `test` blocks are run in parallel with a cap of 3 concurrent processes. The process cap can be set with the `parallel` import argument.

No need to maintain a test count

The test count traditionally was used to ensure your file finished running instead of exiting silently too early. With Test::Builder and friends this has largely been replaced with the `done_testing()` function typically called at the end of tests. Fennec shares this concept, but takes it further, you MUST call `done_testing()` at the end of your test files. This is safer because it can be used to ensure your test script ran completely.

Can be decoupled from Test::Builder

Fennec is built with the assumption that Test::Builder and tools built from it will be used. However custom Fennec::Collector and Fennec::Runner classes can replace this assumption with any testing framework you want to use.

Can run specific test blocks, excluding others

Have you ever had a huge test that took a long time to run? Have you ever needed to debug a failing test at the end of the file? How many times did you need to sit through tests that didn't matter?

With Fennec you can specify the `FENNEC_TEST` environment variable with either a line number or test block name. Only tests defined on that line, or with that name will be run.

Predictability: Rand is always seeded with the date

Randomizing the order in which test blocks are run can help find subtle interaction bugs. At the same time if tests are always in random order you cannot reliably reproduce a failure.

Fennec always seeds rand with the current date. This means that on any given date the test run order will always be the same. However different days test different orders. You can always specify the `FENNEC_SEED` environment variable to override the value used to seed rand.

Diag output is coupled with test output

When you run a Fennec test with a verbose harness (prove −v) the diagnostic output will be coupled with the TAP output. This is done by sending both output to STDOUT. In a non-verbose harness the diagnostics will be sent to STDERR per usual.

Works with Moose

All your test classes are instantiated objects. You can use Moose to define these test classes. But you do not have to, you are not forced to use OOP in your tests.

PROVIDED BY MODULES LOADED BY FENNEC

The 3 most common and useful Test::* modules

Test::More, Test::Warn, Test::Exception

RSPEC support
: Those familiar with Ruby may already know about the RSPEC testing process. In general you `describe` something that is to be tested, then you define setup and teardown methods (`before_all`, `before_each`, `after_all`, `after_each`) and then finally you test `it`. See the ''EXAMPLES'' section or Test::Workflow for more details.

Test re-ordering, tests can run in random, sorted, or defined order.
: When you load Fennec you can specify a test order. The default is random. You can also use the order in which they are defined, or sorted (alphabetically) order. If necessary you can pass in a sorting function that takes a list of all test-objects as arguments.

  Provided by Test::Workflow

Reusable test modules
: You can write tests in modules using Test::Workflow and then import those tests into Fennec tests. This is useful if you have tests that you want run in several, or even all test files.

  Provided by Test::Workflow

Incredibly powerful mocking with a simple API
: You can create classless object instances from a specification on the fly, define new packages, or override existing packages.

  Provided by Mock::Quick

DEFAULT IMPORTED MODULES

Note: These can be overridden either on import, or by subclassing Fennec.

Child − Forking for dummies
: Child is an OO interface to forking that removes all the boilerplate such as checking if the pid changed, and making sure you exit the child process.

Mock::Quick − Mocking without the eye gouging
: Mock::Quick is a mocking library that makes mocking easy. In addition it uses a declarative style interface. Unlike most other mocking libraries on CPAN, it does not make people want to gouge their eyes out and curl up in the fetal position.

Test::Workflow − RSPEC for perl.
: Test::Workflow is a testing library written specifically for Fennec. This library provides RSPEC workflow functions and structure. It can be useful on its own, but combined with Fennec it gets concurrency.

Test::More
: Tried and True testing module that everyone uses.

Test::Warn
: Test::Warn − Test code that issues warnings.

Test::Exception
: Test::Exception − Test code that throws exceptions

IMPORT ARGUMENTS

base => 'Some::Base'
: Load the specified module and make it the base class for your test class.

class => 'What::To::Test'
: Used to specify the name of the package your test file is validating. When this parameter is specified 3 things are done for you: The class is automatically loaded, the `$CLASS` variable is imported and contains the module name, and the *class()* subroutine is defined and returns the name.

```
use Fennec class => 'Foo::Bar';

ok( $INC{'Foo/Bar.pm'}, "Loaded 'Foo::Bar'" );
is( $CLASS, 'Foo::Bar', "We have \$CLASS" );
is( class(), 'Foo::Bar', "We have class()" );

tests method => sub {
```

```
        my $self = shift;
        is( $self->class(), 'Foo::Bar', "We have class() method" );
    };

    done_testing;
```

parallel => $PROC_LIMIT
: How many test blocks can be run in parallel. Default is 3. Set to 1 to fork for each test, but only run one at a time. Set to 0 to prevent forking.

  You can also set this using the $FENNEC_PARALLEL environment variable.

debug => 1
: Enable tracking debugging information. At the end of the Fennec run it will present you with a CSV temp file. This file lists all blocks that are run, and mocks that are made in sequence from top to bottom. The actions are split into columns by PID. This is usedul when debugging potential race-conditions when using parallel testing.

  Example:

```
    26150,26151,26152,26153,26154
    0 26150 BLOCK 54->78 child: outer_wrap, , , , ,
     ,1 26151 BLOCK 47->52 test: class_store, , , ,
    0 26150 MOCK Foo => (outer), , , , ,
    0 26150 BLOCK 58->61 before_all: ba, , , , ,
     , ,2 26152 MOCK Foo => (outer), , ,
     , ,2 26152 BLOCK 63->66 before_each: be, , ,
     , ,2 26152 BLOCK 68->72 test: the_check, , ,
     , , ,3 26153 BLOCK 16->31 test: object, ,
     , , , ,4 26154 BLOCK 33->45 test: class,
```

  You can use this in a spreadsheet program, or use this command to look at it in a more friendly way.

```
    column -s, -t < '/path/to/tempfile' | less -#2 -S
```

collector_class => 'Fennec::Collector::TB::TempFiles'
: Specify which collector to use. Defaults to a Test::Builder based collector that uses temp files to funnel tests from child procs to the parent.

  You generally won't need to specify this, unless you use a test infrastructure that is neither TAP nore Test::Builder based.

runner_class => 'Fennec::Runner'
: Specify the runner class. You probably don't need this.

runner_params => { ... }
: Lets you specify arguments used when Fennec::Runner is initialized.

skip_without => ['Need::This', 'And::This']
: Tell Fennec to skip the test file if any of the specified modules are missing.

test_sort => $SORT
: Options: 'random', 'sorted', 'ordered', or a code block.

  Code block accepts a list of Test::Workflow::Test objects.

utils => ['Test::Foo', ...]
: Load these modules instead of the default list.

  If you need to specify import arguments for any specific util class, you can use the class name as the key with an arrayref containing the arguments.

```
    use Fennec(
        utils          => [ 'Some::Module' ],
        'Some::Module' => [ arg => $val, ... ],
    );
```

with_tests => ['Reusable::Tests', 'Common::Tests']
: Load these modules that have reusable tests. Reusable tests are tests that are common to multiple test files.

seed => '...'
: Set the random seed to be used. Defaults to current date, can be overridden by the FENNEC_SEED environment variable.

debug => `$BOOL`
: Can be used to turn on internal debugging for Fennec. This currently does very little.

ENVIRONMENT VARIABLES

FENNEC_SEED
: Can be used to set a specific random seed

FENNEC_TEST
: Can be used to tell Fennec to only run specific tests (can be given a line number or a block name).

FENNEC_DEBUG
: When true internal debugging is turned on.

EXPORTED FUNCTIONS

FROM FENNEC

done_testing()

done_testing(sub { ... })
: Should be called at the end of your test file to kick off the RSPEC tests. Always returns 1, so you can use it as the last statement of your module. You must only ever call this once per test file.

  Never put tests below the done_testing call. If you want tests to run AFTER the RSPEC workflow completes, you can pass done_testing a coderef with the tests.

```
done_testing( sub {
    ok( 1, "This runs after the RSPEC workflow" );
});
```

FROM Test::Workflow

See Test::Workflow or "EXAMPLES" for more details.

with_tests 'Module::Name';
: Import tests from a module

tests `$name` => sub { ... };
tests `$name` => (`%params`);
it `$name` => sub { ... };
it `$name` => (`%params`);
: Define a test block

describe `$name` => sub { ... };
: Describe a set of tests (group tests and setup/teardown functions)

case `$name` => sub { ... };
: Used to run a set of tests against multiple conditions

before_all `$name` => sub { ... };
: Setup, run once before any tests in the describe scope run.

before_case `$name` => sub { ... };
: Setup, run before any case blocks are run.

before_each `$name` => sub { ... };
after_case `$name` => sub { ... };
: Setup, run once per test, just before it runs. Both run after the case block (if there is one).

around_each `$name` => sub { ... };
: Setup and/or teardown.

after_each `$name` => sub { ... };
: Teardown, run once per test, after it finishes.

after_all $name => sub { ... };
Teardown, run once, after all tests in the describe scope complete.

FROM Mock::Quick

See Mock::Quick or ''EXAMPLES'' for more details.

my $control = qclass $CLASS => (%PARAMS, %OVERRIDES);
my $control = qtakeover $CLASS => (%PARAMS, %OVERRIDES);
my $control = qimplement $CLASS => (%PARAMS, %OVERRIDES);
my $control = qcontrol $CLASS => (%PARAMS, %OVERRIDES);
Used to define, takeover, or override parts of other packages.

my $obj = qobj(%PARAMS);
my ($obj, $control) = qobjc(%PARAMS);
my $obj = qstrict(%PARAMS);
my ($obj, $control) = qstrictc(%PARAMS);
Define an object specification, quickly.

my $clear = *qclear()*;
Used to clear a field in a quick object.

my $meth = qmeth { ... };
my $meth = qmeth(sub { ... });
Used to define a method for a quick object.

OTHER

See Test::More, Test::Warn, and Test::Exception

EXAMPLES

Examples can be the best form of documentation.

SIMPLE

VANILLA SYNTAX

t/simple.t

```
use strict;
use warnings;

use Fennec;

use_ok 'Data::Dumper';

tests dumper => sub {
    my $VAR1;
    is_deeply(
        eval Dumper({ a => 1 }),
        { a => 1 },
        "Serialize and De-Serialize"
    );
};

tests future => (
    todo => "Not ready yet",
    code => sub {
        ok( 0, "I still have to write these" );
    },
);

done_testing;
```

DECLARE SYNTAX

t/simple.t

```
use strict;
use warnings;

use Fennec::Declare;

use_ok 'Data::Dumper';

tests dumper {
    my $VAR1;
    is_deeply(
        eval Dumper({ a => 1 }),
        { a => 1 },
        "Serialize and De-Serialize"
    );

    is(
        eval { no strict; Dumper( { a => 1 } ) },
        { a => 1 },
        "Serialize and De-Serialize"
    );
}

tests future( todo => "Not ready yet" ) {
    ok( 0, "I still have to write these" );
}

done_testing;
```

RUN TESTS UNDER DIFFERENT CONDITIONS

This example shows 4 conditions (`$letter` as 'a', 'b', 'c', and 'd'). It also has 2 test blocks, one that verifies `$letter` is a letter, the other verifies it is lowercase. Each test block will be run once for each condition, 2*4=8, so in total 8 tests will be run.

VANILLA

sample.t:

```
use strict;
use warnings;

use Fennec;

my $letter;
case a => sub { $letter = 'a' };
case b => sub { $letter = 'b' };
case c => sub { $letter = 'c' };
case d => sub { $letter = 'd' };

tests is_letter => sub {
    like( $letter, qr/^[a-z]$/i, "Got a letter" );
};

tests is_lowercase => sub {
    is( $letter, lc( $letter ), "Letter is lowercase" );
};

done_testing;
```

OBJECT ORIENTED

sample.t

```
    use strict;
    use warnings;

    use Fennec;

    sub letter {
        my $self = shift;
        ( $self->{letter} ) = @_ if @_;
        return $self->{letter};
    }

    describe letters => sub {
        case a => sub { shift->letter('a') };
        case b => sub { shift->letter('b') };
        case c => sub { shift->letter('c') };
        case d => sub { shift->letter('d') };

        tests is_letter => sub {
            my $self = shift;
            like( $self->letter, qr/^[a-z]$/i, "Got a letter" );
        };

        tests is_lowercase => sub {
            my $self = shift;
            is( $self->letter, lc( $self->letter ), "Letter is lowercase" );
        };
    };

    done_testing;
```

DECLARE

Note: no need to shift `$self`, it is done for you!

sample.t

```
    use strict;
    use warnings;

    use Fennec::Declare;

    sub letter {
        my $self = shift;
        ( $self->{letter} ) = @_ if @_;
        return $self->{letter};
    }

    describe letters {
        case a { $self->letter('a') }

        case b { $self->letter('b') }

        case c { $self->letter('c') }

        case d { $self->letter('d') }

        tests is_letter {
            like( $self->letter, qr/^[a-z]$/i, "Got a letter" );
        }

        tests is_lowercase {
```

```
            is( $self->letter, lc( $self->letter ), "Letter is lowercase" );
        }
    }

    done_testing;
```

MOCKING

See Mock::Quick for more details

OBJECT ON THE FLY

```
    my $obj = qobj(
        foo => 'foo',
        bar => qmeth { 'bar' },
        baz => sub { 'baz' },
    );

    is( $obj->foo, 'foo' );
    is( $obj->bar, 'bar' );
    is( ref $obj->baz, 'CODE', "baz is a method that returns a coderef" );

    # All methods autovivify as read/write accessors:
    lives_ok { $obj->blah( 'x' ) };

    # use qstrict() to make an object that does not autovivify accessors.
```

SCOPE OF MOCKS IN FENNEC

With vanilla Mock::Quick a mock is destroyed when the control object is destroyed.

```
    my $control = qtakeover Foo => (blah => 'blah');
    is( Foo->blah, 'blah', "got mock" );
    $control = undef;
    ok( !Foo->can('blah'), "Mock destroyed" );

    # WITHOUT FENNEC This issues a warning, the $control object is ignored so
    # the mock is destroyed before it can be used.
    qtakover Foo => (blah => 'blah');
    ok( !Foo->can('blah'), "Mock destroyed before it could be used" );
```

With the workflow support provided by Fennec, you can omit the control object and let the mock be scoped implicitly.

```
    tests implicit_mock_scope => sub {
        my $self = shift;
        can_ok( $self, 'QINTERCEPT' );
        qtakeover Foo => (blah => sub { 'blah' });
        is( Foo->blah, 'blah', "Mock not auto-destroyed" );
    };

    describe detailed_implicit_mock_scope => sub {
        qtakeover Foo => ( outer => 'outer' );
        ok( !Foo->can( 'outer' ), "No Leak" );

        before_all ba => sub {
            qtakeover Foo => ( ba => 'ba' );
            can_ok( 'Foo', qw/outer ba/ );
        };

        before_each be => sub {
            qtakeover Foo => ( be => 'be' );
            can_ok( 'Foo', qw/outer ba be/ );
        };
```

```
        tests the_check => sub {
            qtakeover Foo => ( inner => 'inner' );

            can_ok( 'Foo', qw/outer ba be inner/ );
        };

        ok( !Foo->can( 'outer' ), "No Leak" );
        ok( !Foo->can( 'ba' ), "No Leak" );
        ok( !Foo->can( 'be' ), "No Leak" );
        ok( !Foo->can( 'inner' ), "No Leak" );
    };
```

TAKEOVER AN EXISTING CLASS

```
    require Some::Class;
    my $control = qtakeover 'Some::Class' => (
        # Override some methods:
        foo => sub { 'foo' },
        bar => sub { 'bar' },

        # For methods that return a simple value you don't actually need to
        # wrap them in a sub.
        baz => 'bat',
    );

    is( Some::Class->foo, 'foo' );
    is( Some::Class->bar, 'bar' );

    # Use the control object to make another override
    $control->override( foo => 'FOO' );
    is( Some::Class->foo, 'FOO' );

    # Original class is restored when $control falls out of scope.
    $control = undef;
```

MOCK A CLASS INSTEAD OF LOADING THE REAL ONE

This will prevent the real class from loading if code tries to `require` or `use` it. However when the control object falls out of scope you will be able to load the real one again.

```
    my $control = qimplement 'Some::Class' => (
        my_method => sub { ... }
        simple    => 'foo',
    );
```

MOCK AN ANONYMOUS CLASS

```
    my $control = qclass(
        -with_new => 1, # Make a constructor for us
        method => sub { ... },
        simple => 'foo',
    );

    my $obj = $control->package->new;
```

REUSABLE TEST LIBRARIES

This is a test library that verifies your test file uses strict in the first 3 lines. You can also pass `with_tests => [ 'Some::Test::Lib' ]` as an import argument to Fennec. This matters because you can subclass Fennec to always include this library.

t/test.t

```
use strict;
use warnings;
use Fennec;

with_tests 'Some::Test::Lib';

done_testing;
```

lib/Some/Test/Lib.pm

```
package Some::Test::Lib;
use Test::Workflow;
use Test::More;
use Scalar::Util qw/blessed/;

tests check_use_strict => sub {
    my $self  = shift;
    my $class = blessed $self;

    my $file = $class;
    $file =~ s{::}{/}g;
    $file .= '.pm';

    my $full = $INC{$file};
    ok( -e $full, "Found path and filename for $class" );
    open( my $fh, '<', $full ) || die $!;
    my $found = 0;

    for ( 1 .. 3 ) {
        $found = <$fh> =~ m/^\s*use strict;\s*$/;
        last if $found;
    }
    close($fh);
    ok( $found, "'use strict;' is in the first 3 lines of the test file" );
}

1;
```

POST TESTS

You cannot put any tests under `done_testing()` Doing so will cause problems. However you can put tests IN done_testing.

```
use strict;
use warnings;

use Fennec;

my $foo = 1;

is( $foo, 1, "foo is 1" );

done_testing(
    sub {
        is( $foo, 1, "foo is still 1" );
    }
);
```

RSPEC

The following test will produce output similar to the following. Keep in mind that the concurrent nature of Fennec means that the lines for each process may appear out of order relative to lines from other processes. Lines for any given process will always be in the correct order though.

Spacing has been added, and process output has been grouped together, except for the main process to demonstrate that after_all really does come last.

```
# PID          OUTPUT
#-------------------------------------------
7253 describe runs long before everything else
7253 before_all runs first

7254 Case runs between before_all and before_each
7254 before_each runs just before tests
7254 tests run in the middle
7254 after_each runs just after tests

7255 before_each runs just before tests
7255 This test inherits the before and after blocks from the parent describe.
7255 after_each runs just after tests

7253 after_all runs last.
```

sample.t

```
use strict;
use warnings;

use Fennec;

describe order => sub {
    print "$$ describe runs long before everything else\n";

    before_all setup_a => sub {
        print "$$ before_all runs first\n";
    };

    case a_case => sub {
        print "$$ Case runs between before_all and before_each\n";
    };

    before_each setup_b => sub {
        print "$$ before_each runs just before tests\n";
    };

    tests a_test => sub {
        print "$$ tests run in the middle\n";
    };

    after_each teardown_b => sub {
        print "$$ after_each runs just after tests\n";
    };

    after_all teardown_a => sub {
        print "$$ after_all runs last.\n";
    };

    describe nested => sub {
        tests b_test => sub {
            print "$$ This test inherits the before/after/case blocks from th
        };
    };
};
```

```
done_testing;
```

MANUAL

The manual can be found here: Fennec::Manual it is a sort of Nexus for documentation, including this document.

VIM INTEGRATION

Insert this into your .vimrc file to bind the F8 key to running the test block directly under your cursor. You can be on any line of the test block (except in some cases the first or last line.

```
function! RunFennecLine()
    let cur_line = line(".")
    exe "!FENNEC_TEST='" . cur_line . "' prove -v -I lib %"
endfunction

" Go to command mode, save the file, run the current test
:map <F8> <ESC>:w<cr>:call RunFennecLine()<cr>
:imap <F8> <ESC>:w<cr>:call RunFennecLine()<cr>
```

RUNNING FENNEC TEST FILES IN PARALLEL

The best option is to use prove with the -j flag.

Note: The following is no longer a recommended practice, it is however still supported

You can also create a custom runner using a single .t file to run all your Fennec tests. This has caveats though, such as not knowing which test file had problems without checking the failure messages.

This will find all *.ft and/or *.pm modules under the t/ directory. It will load and run any found. These will be run in parallel

t/runner.t

```
#!/usr/bin/perl
use strict;
use warnings;

# Paths are optional, if none are specified it defaults to 't/'
use Fennec::Finder( 't/' );

# The next lines are optional, if you have no custom configuration to apply
# you can jump right to 'done_testing'.

# Get the runner (singleton)
my $runner = Fennec::Finder->new;
$runner->parallel( 3 );

# You must call this.
run();
```

AUTHORS

Chad Granum exodist7 AT gmail DOT com

COPYRIGHT