NAME

Dancer::Exception − class for throwing and catching exceptions

VERSION

version 1.3140

SYNOPSIS

```
use Dancer::Exception qw(:all);

register_exception('DataProblem',
                   message_pattern => "test message : %s"
                  );

sub do_stuff {
  raise DataProblem => "we've lost data!";
}

try {
  do_stuff()
} catch {
  # an exception was thrown
  my ($exception) = @_;
  if ($exception->does('DataProblem')) {
    # handle the data problem
    my $message = $exception->message();
  } else {
    $exception->rethrow
  }
};
```

DESCRIPTION

Dancer::Exception is based on Try::Tiny. You can try and catch exceptions, like in Try::Tiny.

Exceptions are objects, from subclasses of Dancer::Exception::Base.

However, for internal Dancer usage, we introduce a special class of exceptions, called Dancer::Continuation. Exceptions that are from this class are not caught with a `catch` block, but only with a `continuation`. That's a cheap way to implement a *workflow interruption*. Dancer users should ignore this feature.

What it means for Dancer users

Users can throw and catch exceptions, using `try` and `catch`. They can reuse some Dancer core exceptions (`Dancer::Exception::Base::*`), but they can also create new exception classes, and use them for their own means. That way it's easy to use custom exceptions in a Dancer application. Have a look at `register_exception`, `raise`, and the methods in Dancer::Exception::Base.

METHODS

try

Same as in Try::Tiny

catch

Same as in Try::Tiny. The exception can be retrieved as the first parameter:

```
try { ... } catch { my ($exception) = @_; };
```

continuation

To be used by Dancer developers only, in Dancer core code.

raise

```
# raise Dancer::Exception::Base::Custom
raise Custom => "user $username is unknown";

# raise Dancer::Exception::Base::Custom::Frontend
raise 'Custom::Frontend' => "user $username is unknown";
```

```
# same, raise Dancer::Exception::Base::Custom::Frontend
raise custom_frontend => "user $username is unknown";

# raise My::Own::ExceptionSystem::Invalid::Login
raise '+My::Own::ExceptionSystem::Invalid::Login' => "user $username is unknown
```

raise provides an easy way to throw an exception. First parameter is the name of the exception class, without the `Dancer::Exception::` prefix. other parameters are stored as *raising arguments* in the exception. Usually the parameters is an exception message, but it's left to the exception class implementation.

If the exception class name starts with a `+`, then the `Dancer::Exception::` won't be added. This allows one to build their own exception class hierarchy, but you should first look at `register_exception` before implementing your own class hierarchy. If you really wish to build your own exception class hierarchy, we recommend that all exceptions inherit of Dancer::Exception::. Or at least it should implement its methods.

The exception class can also be written as words separated by underscores, it'll be camelized automatically. So `'Exception::Foo'` and `'exception_foo'` are equivalent. Be careful, `'MyException'` can't be written `'myexception'`, as it would be camelized into `'Myexception'`.

register_exception

This method allows one to register custom exceptions, usable by Dancer users in their route code (actually pretty much everywhere).

```
# simple exception
register_exception ('InvalidCredentials',
                    message_pattern => "invalid credentials : %s",
                   );
```

This registers a new custom exception. To use it, do:

```
raise InvalidCredentials => "user Herbert not found";
```

The exception message can be retrieved with the `$exception->message` method, and we'll be `"invalid credentials : user Herbert not found"` (see methods in Dancer::Exception::Base)

```
# complex exception
register_exception ('InvalidLogin',
                    composed_from => [qw(Fatal InvalidCredentials)],
                    message_pattern => "wrong login or password",
                  );
```

In this example, the `InvalidLogin` is built as a composition of the `Fatal` and `InvalidCredentials` exceptions. See the `does` method in Dancer::Exception::Base.

registered_exceptions

```
my @exception_classes = registered_exceptions;
```

Returns the list of exception class names. It will list core exceptions `and` custom exceptions (except the one you've registered with a leading `+`, see `register_exception`). The list is sorted.

GLOBAL VARIABLE

`$Dancer::Exception::Verbose`

When set to 1, exceptions will stringify with a long stack trace. This variable is similar to `$Carp::Verbose`. I recommend you use it like that:

```
local $Dancer::Exception::Verbose;
$Dancer::Exception::Verbose = 1;
```

All the Carp global variables can also be used to alter the stacktrace generation.

AUTHOR

Dancer Core Developers

COPYRIGHT AND LICENSE