NAME

Dancer2::Cookbook − Example−driven quick−start to the Dancer2 web framework

VERSION

version 0.160003

DESCRIPTION

A quick-start guide with examples to get you up and running with the Dancer2 web framework. This document will be twice as useful if you finish reading the manual (Dancer2::Manual) first, but that is not required... :−)

BEGINNER'S DANCE

A simple Dancer2 web app

Dancer2 has been designed to be easy to work with − it's trivial to write a simple web app, but still has the power to work with larger projects. To start with, let's make an incredibly simple ''Hello World'' example:

```
#!/usr/bin/env perl

use Dancer2;

get '/hello/:name' => sub {
    return "Why, hello there " . params->{name};
};

dance;
```

Yes − the above is a fully-functioning web app; running that script will launch a webserver listening on the default port (3000). Now you can make a request:

```
$ curl http://localhost:3000/hello/Bob
Why, hello there Bob
```

and it will say hello. The `:name` part is a named parameter within the route specification, whose value is made available through `params`.

Note that you don't need to use the `strict` and `warnings` pragmas; they are already loaded by Dancer2.

Default Route

In case you want to avoid a *404 error*, or handle multiple routes in the same way and you don't feel like configuring all of them, you can set up a default route handler.

The default route handler will handle any request that doesn't get served by any other route.

All you need to do is set up the following route as the **last** route:

```
any qr{.*} => sub {
    status 'not_found';
    template 'special_404', { path => request->path };
};
```

Then you can set up the template like so:

```
You tried to reach [% path %], but it is unavailable at the moment.

Please try again or contact us at <contact AT example DOT com>.
```

Using the `auto_page` feature for automatic route creation

For simple ''static'' pages you can simply enable the `auto_page` config setting; this means you don't need to declare a route handler for those pages; if a request is for `/foo/bar`, Dancer2 will check for a matching view (e.g. `/foo/bar.tt` and render it with the default layout, if found. For full details, see the documentation for the auto_page setting.

Simplifying AJAX queries with the Ajax plugin

As an AJAX query is just an HTTP query, it's similar to a GET or POST route. You may ask yourself why you may want to use the `ajax` keyword (from the Dancer2::Plugin::Ajax plugin) instead of a simple `get`.

Let's say you have a path like `/user/:user` in your application. You may want to be able to serve this page with a layout and HTML content. But you may also want to be able to call this same url from a javascript query using AJAX.

So, instead of having the following code:

```
get '/user/:user' => sub {
     if ( request->is_ajax ) {
         # create xml, set headers to text/xml, blablabla
          header( 'Content-Type'  => 'text/xml' );
          header( 'Cache-Control' =>  'no-store, no-cache, must-revalidate' )
          to_xml({...})
     } else {
         template users => {...}
     }
};
```

you can have

```
ajax '/user/:user' => sub {
     to_xml( {...}, RootName => undef );
}
```

and

```
get '/user/:user' => sub {
    template users => {...}
}
```

Because it's an AJAX query, you know you need to return XML content, so the content type of the response is set for you.

Example: Feeding graph data through AJAX

Let us assume we are building an application that uses a plotting library to generate a graph and expects to get its data, which is in the form of wordcount from an AJAX call.

For the graph, we need the url */data* to return a JSON representation of the wordcount data. Dancer infact has a `to_json()` function that takes care of the JSON encapsulation.

```
 get '/data' => sub {
     open my $fh, '<', $count_file;

     my %contestant;
     while (<$fh>) {
         chomp;
         my ( $date, $who, $count ) = split '\s*,\s*';

         my $epoch = DateTime::Format::Flexible->parse_datetime($date)->epoch
         my $time = 1000 * $epoch;
         $contestant{$who}{$time} = $count;
     }

     my @json;  # data structure that is going to be JSONified

     while ( my ( $peep, $data ) = each %contestant ) {
         push @json, {
             label     => $peep,
             hoverable => \1,    # so that it becomes JavaScript's 'true'
             data => [ map  { [ $_, $data->{$_} ] }
                     sort { $a <=> $b }
                     keys %$data ],
         };
     }
```

```
        my $beginning = DateTime::Format::Flexible->parse_datetime( "2010-11-01"
        my $end       = DateTime::Format::Flexible->parse_datetime( "2010-12-01"

        push @json, {
            label => 'de par',
            data => [
                [$beginning * 1000, 0],
                [   DateTime->now->epoch * 1_000,
                    50_000
                      * (DateTime->now->epoch - $beginning)
                      / ($end - $beginning)
                ]
              ],

        };

        to_json( \@json );
    };
```

For more serious AJAX interaction, there's also Dancer2::Plugin::Ajax that adds an *ajax* route handler to the mix.

Because it's an AJAX query, you know you need to return XML content, so the content type of the response is set for you.

Using the prefix feature to split your application

For better maintainability, you may want to separate some of your application components into different packages. Let's say we have a simple web app with an admin section and want to maintain this in a different package:

```
package myapp;
use Dancer2;
use myapp::admin;

prefix undef;

get '/' => sub {...};

1;

package myapp::admin;
use Dancer2 appname => 'myapp';

prefix '/admin';

get '/' => sub {...};

1;
```

The following routes will be generated for us:

```
- get /
- get /admin/
- head /
- head /admin/
```

By default, a separate application is created for every package that uses Dancer2. The `appname` tag is used to collect routes and hooks into a single Dancer2 application. In the above example, `appname => 'myapp'` adds the routes from `myapp::admin` to the routes of the app `myapp`.

When using multiple applications please ensure that your path definitions do not overlap. For example, if using a default route as described above, once a request is matched to the default route then no further routes (or applications) would be reached.

Delivering custom error pages

At the Core

In Dancer2, creating new errors is done by creating a new Dancer2::Core::Error

```
my $oopsie = Dancer2::Core::Error->new(
    status  => 418,
    message => "This is the Holidays. Tea not acceptable. We want eggnog.",
    app     => $app,
)
```

If not given, the status code defaults to a 500, there is no need for a message if we feel taciturn, and while the `$app` (which is a *Dancer::Core::App* object holding all the pieces of information related to the current request) is needed if we want to take advantage of the templates, we can also do without.

However, to be seen by the end user, we have to populate the Dancer2::Core::Response object with the error's data. This is done via:

```
$oopsie->throw($response);
```

Or, if we want to use the response object already present in the `$app` (which is usually the case):

```
$oopsie->throw;
```

This populates the status code of the response, sets its content, and throws a *halt()* in the dispatch process.

What it will look like

The error object has quite a few ways to generate its content.

First, it can be explicitly given

```
my $oopsie = Dancer::Core::Error->new(
    content => '<html><body><h1>OMG</h1></body></html>',
);
```

If the `$context` was given, the error will check if there is a template by the name of the status code (so, say you're using Template Toolkit, *418.tt*) and will use it to generate the content, passing it the error's `$message`, `$status` code and `$title` (which, if not specified, will be the standard http error definition for the status code).

If there is no template, the error will then look for a static page (to continue with our example, *418.html*) in the *public/* directory.

And finally, if all of that failed, the error object will fall back on an internal template.

Errors in Routes

The simplest way to use errors in routes is:

```
get '/xmas/gift/:gift' => sub {
    die "sorry, we're all out of ponies\n"
        if param('gift') eq 'pony';
};
```

The die will be intercepted by Dancer, converted into an error (status code 500, message set to the dying words) and passed to the response.

In the cases where more control is required, `send_error()` is the way to go:

```
get '/glass/eggnog' => sub {
    send_error "Sorry, no eggnog here", 418;
};
```

And if total control is needed:

```
get '/xmas/wishlist' => sub {
    Dancer::Core::Error->new(
        response => response(),
        status   => 406,
        message  => "nothing but coal for you, I'm afraid",
        template => 'naughty/index',
    )->throw unless user_was_nice();

    ...;
};
```

Template Toolkit's WRAPPER directive in Dancer2

Dancer2 already provides a WRAPPER-like ability, which we call a ''layout''. The reason we don't use Template Toolkit's WRAPPER (which also makes us incompatible with it) is because not all template systems support it. In fact, most don't.

However, you might want to use it, and be able to define META variables and regular Template::Toolkit variables.

These few steps will get you there:

- Disable the layout in Dancer2

  You can do this by simply commenting (or removing) the `layout` configuration in the config file.

- Use the Template Toolkit template engine

  Change the configuration of the template to Template Toolkit:

  ```
  # in config.yml
  template: "template_toolkit"
  ```

- Tell the Template Toolkit engine which wrapper to use

  ```
  # in config.yml
  # ...
  engines:
      template:
          template_toolkit:
              WRAPPER: layouts/main.tt
  ```

Done! Everything will work fine out of the box, including variables and META variables.

Accessing configuration information from a separate script

You may want to access your webapp's configuration from outside your webapp. You could, of course, use the YAML module of your choice and load your webapps's `config.yml`, but chances are that this is not convenient.

Use Dancer2 instead. You can simply use the values from `config.yml` and some additional default values:

```
# bin/show_app_config.pl
use Dancer2;
printf "template: %s\n", config->{'template'}; # simple
printf "log: %s\n",      config->{'log'};      # undef
```

Note that `config->{log}` should result in an uninitialized warning on a default scaffold since the environment isn't loaded and log is defined in the environment and not in `config.yml`. Hence `undef`.

Dancer2 will load your `config.yml` configuration file along with the correct environment file located in your `environments` directory.

The environment is determined by two environment variables in the following order:

- DANCER_ENVIRONMENT
- PLACK_ENV

If neither of those is set, it will default to loading the development environment (typically

`$webapp/environment/development.yml`).

If you wish to load a different environment, you need to override these variables.

You can call your script with the environment changed:

```
$ PLACK_ENV=production perl bin/show_app_config.pl
```

Or you can override them directly in the script (less recommended):

```
BEGIN { $ENV{'DANCER_ENVIRONMENT'} = 'production' }
use Dancer2;

...
```

Using DBIx::Class

DBIx::Class, also known as DBIC, is one of the many Perl ORM (*Object Relational Mapper*). It is easy to use DBIC in Dancer2 using the Dancer2::Plugin::DBIC.

An example

This example demonstrates a simple Dancer2 application that allows one to search for authors or books. The application is connected to a database, that contains authors, and their books. The website will have one single page with a form, that allows one to query books or authors, and display the results.

Creating the application

```
$ dancer2 −a bookstore
```

To use the Template Toolkit as the template engine, we specify it in the congiguration file:

```
# add in bookstore/config.yml
template: template_toolkit
```

Creating the view

We need a view to display the search form, and below, the results, if any. The results will be fed by the route to the view as an arrayref of results. Each result is a *hashref*, with a author key containing the name of the author, and a books key containing an *arrayref* of strings : the books names.

```
# example of a list of results
[ { author => 'author 1',
    books => [ 'book 1', 'book 2' ],
  },
  { author => 'author 2',
    books => [ 'book 3', 'book 4' ],
  }
]
```

bookstore/views/search.tt <form action=‘‘/search’’> Search query: <input type=‘‘text’’ name=‘‘query’’ /> </form>

An example of the view, displaying the search form, and the results, if any:

```
<% IF query.length %>
  <p>Search query was : <% query %>.</p>
  <% IF results.size %>
    Results:
    <ul>
    <% FOREACH result IN results %>
      <li>Author: <% result.author.replace("((?i)$query)", '<b>$1</b>') %>
      <ul>
      <% FOREACH book IN result.books %>
        <li><% book.replace("((?i)$query)", '<b>$1</b>') %>
      <% END %>
      </ul>
    <% END %>
  <% ELSE %>
```

```
    No result
  <% END %>
<% END %>
```

Creating a Route

A simple route, to be added in the *bookstore.pm* module:

```
# add in bookstore/lib/bookstore.pm
get '/search' => sub {
    my $query   = params->{'query'};
    my @results = ();

    if ( length $query ) {
        @results = _perform_search($query);
    }

    template search => {
        query   => $query,
        results => \@results,
    };
};
```

Creating a database

We create a SQLite file database:

```
$ sqlite3 bookstore.db
CREATE TABLE author(
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  firstname text default '' not null,
  lastname text not null);

CREATE TABLE book(
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  author INTEGER REFERENCES author (id),
  title text default '' not null );
```

Now, to populate the database with some data, we use DBIx::Class:

```
# populate_database.pl
package My::Bookstore::Schema;
use base qw(DBIx::Class::Schema::Loader);
package main;
my $schema = My::Bookstore::Schema->connect('dbi:SQLite:dbname=bookstore.db')
$schema->populate('Author', [
  [ 'firstname', 'lastname'],
  [ 'Ian M.',    'Banks'   ],
  [ 'Richard',   'Matheson'],
  [ 'Frank',     'Herbert' ],
]);
my @books_list = (
  [ 'Consider Phlebas',    'Banks'    ],
  [ 'The Player of Games', 'Banks'    ],
  [ 'Use of Weapons',      'Banks'    ],
  [ 'Dune',                'Herbert'  ],
  [ 'Dune Messiah',        'Herbert'  ],
  [ 'Children of Dune',    'Herbert'  ],
  [ 'The Night Stalker',   'Matheson' ],
  [ 'The Night Strangler', 'Matheson' ],
);
# transform author names into ids
$_->[1] = $schema->resultset('Author')->find({ lastname => $_->[1] })->id
```

```
  foreach (@books_list);
$schema->populate('Book', [
  [ 'title', 'author' ],
  @books_list,
]);
```

Then run it in the directory where *bookstore.db* sits:

```
perl populate_database.db
```

Using Dancer2::Plugin::DBIC

There are 2 ways of configuring DBIC to understand how the data is organized in your database:

- Use auto-detection

  The configuration file needs to be updated to indicate the use of the Dancer2::Plugin::DBIC plugin, define a new DBIC schema called *bookstore* and to indicate that this schema is connected to the SQLite database we created.

  ```
  # add in bookstore/config.yml
  plugins:
    DBIC:
      bookstore:
        dsn:  "dbi:SQLite:dbname=bookstore.db"
  ```

  Now, `_perform_search` can be implemented using Dancer2::Plugin::DBIC. The plugin gives you access to an additional keyword called **schema**, which you give the name of schema you want to retrieve. It returns a `DBIx::Class::Schema::Loader` which can be used to get a resultset and perform searches, as per standard usage of DBIX::Class.

  ```
  # add in bookstore/lib/bookstore.pm
  sub _perform_search {
      my ($query) = @_;
      my $bookstore_schema = schema 'bookstore';
      my @results;
      # search in authors
      my @authors = $bookstore_schema->resultset('Author')->search({
        -or => [
          firstname => { like => "%$query%" },
          lastname  => { like => "%$query%" },
        ]
      });
      push @results, map {
          { author => join(' ', $_->firstname, $_->lastname),
            books => [],
          }
      } @authors;
      my %book_results;
      # search in books
      my @books = $bookstore_schema->resultset('Book')->search({
          title => { like => "%$query%" },
      });
      foreach my $book (@books) {
          my $author_name = join(' ', $book->author->firstname, $book->autho
          push @{$book_results{$author_name}}, $book->title;
      }
      push @results, map {
          { author => $_,
            books => $book_results{$_},
          }
      } keys %book_results;
      return @results;
  }
  ```

- Use home made schema classes

  The DBIx::Class::MooseColumns lets you write the DBIC schema classes using Moose. The schema classes should be put in a place that Dancer2 will find. A good place is in *bookstore/lib/*.

  Once your schema classes are in place, all you need to do is modify *config.yml* to specify that you want to use them, instead of the default auto-detection method:

```
# change in bookstore/config.yml
plugins:
  DBIC:
    bookstore:
      schema_class: My::Bookstore::Schema
      dsn: "dbi:SQLite:dbname=bookstore.db"
```

  Starting the application: Our bookstore lookup application can now be started using the built-in server:

```
# start the web application
bookstore/bin/app.pl
```

Authentication

Writing a form for authentication is simple: we check the user credentials on a request and decide whether to continue or redirect them to a form. The form allows them to submit their username and password and we save that and create a session for them so when they now try the original request, we recognize them and allow them in.

Basic Application

The application is fairly simple. We have a route that needs authentication, we have a route for showing the login page, and we have a route for posting login information and creating a session.

```
package MyApp;
use Dancer2;

get '/' => sub {
    session('user')
        or redirect('/login');

    template index => {};
};

get '/login' => sub {
    template login => {};
};

post '/login' => sub {
    my $username  = param('username');
    my $password  = param('password');
    my $redir_url = param('redirect_url') || '/login';

    $username eq 'john' && $password eq 'correcthorsebatterystaple'
        or redirect $redir_url;

    session user => $username;
    redirect $redir_url;
};
```

Tiny Authentication Helper

Dancer2::Plugin::Auth::Tiny allows you to abstract away not only the part that checks whether the session exists, but to also generate a redirect with the right path and return URL.

We simply have to define what routes needs a login using Auth::Tiny's `needs` keyword.

```
get '/' => needs login => sub {
    template index => {};
};
```

It creates a proper return URL using `uri_for` and the address from which the user arrived.

We can thus decorate all of our private routes to require authentication in this manner. If a user does not have a session, it will automatically forward it to */login*, in which we would render a form for the user to send a login request.

Auth::Tiny even provides a new parameter, `return_url`, which can be used to send the user back to their original requested path.

Password Hashing

Dancer2::Plugin::Passphrase provides a simple passwords-as-objects interface with sane defaults for hashed passwords which you can use in your web application. It uses **bcrypt** as the default but supports anything the Digest interface does.

Assuming we have the original user-creation form submitting a username and password:

```
package MyApp;
use Dancer2;
use Dancer2::Plugin::Passphrase;
post '/register' => sub {
    my $username = param('username');
    my $password = passphrase( param('password') )->generate;

    # $password is now a hashed password object
    save_user_in_db( $username, $password->rfc2307 );

    template registered => { success => 1 };
};
```

We can now add the **POST** method for verifying that username and password:

```
post '/login' => sub {
    my $username   = param('username');
    my $password   = param('password');
    my $saved_pass = fetch_password_from_db($username);

    if ( passphrase($password)->matches($saved_pass) ) {
        session user => $username;
        redirect param('return_url') || '/';
    }

    # let's render instead of redirect...
    template login => { error => 'Invalid username or password' };
};
```

Writing a REST application

With Dancer2, it's easy to write REST applications. Dancer2 provides helpers to serialize and deserialize for the following data formats:

JSON
YAML
XML
Data::Dumper

To activate this feature, you only have to set the `serializer` setting to the format you require, for instance in your config file:

```
serializer: JSON
```

Or directly in your code:

```
set serializer => 'JSON';
```

From now, all hashrefs or arrayrefs returned by a route will be serialized to the format you chose, and all data received from **POST** or **PUT** requests will be automatically deserialized.

```
get '/hello/:name' => sub {
    # this structure will be returned to the client as
    # {"name":"$name"}
    return {name => params->{name}};
};
```

It's possible to let the client choose which serializer to use. For this, use the `mutable` serializer, and an appropriate serializer will be chosen from the `Content-Type` header.

It's also possible to return a custom error using the send_error keyword. When you don't use a serializer, the `send_error` function will take a string as first parameter (the message), and an optional HTTP code. When using a serializer, the message can be a string, an arrayref or a hashref:

```
get '/hello/:name' => sub {
    if (...) {
       send_error("you can't do that");
       # or
       send_error({reason => 'access denied', message => "no"});
    }
};
```

The content of the error will be serialized using the appropriate serializer.

Using the serializer

Serializers essentially do two things:

- Deserialize incoming requests

  When a user makes a request with serialized input, the serializer automatically deserializes it into actual input parameters.

- Serialize outgoing responses

  When you return a data structure from a route, it will automatically serialize it for you before returning it to the user.

Configuring

In order to configure a serializer, you just need to pick which format you want for encoding/decoding (from Dancer2::Serializer) and set it up using the `serializer` configuration keyword.

It is recommended to explicitly add it in the actual code instead of the configuration file so it doesn't apply automatically to every app that reads the configuration file (unless that's what you want):

```
package MyApp;
use Dancer2;
set serializer => 'JSON'; # Dancer2::Serializer::JSON

...
```

Using

Now that we have a serializer set up, we can just return data structures:

```
get '/' => sub {
    return { resources => \%resources };
};
```

When we return this data structure, it will automatically be serialized into JSON. No other code is necessary.

We also now receive requests in JSON:

```
post '/:entity/:id' => sub {
    my $entity = param('entity');
    my $id     = param('id');
```

```
    # input which was sent serialized
    my $user = param('user');

    ...
};
```

We can now make a serialized request:

```
$ curl -X POST http://ourdomain/person/16 -d '{"user":"sawyer_x"}'
```

App-specific feature

Serializers are engines. They affect a Dancer Application, which means that once you've set a serializer, **all** routes within that package will be serialized and deserialized. This is how the feature works.

As suggested above, if you would like to have both, you need to create another application which will not be serialized.

A common usage for this is an API providing serialized endpoints (and receiving serialized requests) and providing rendered pages.

```
# MyApp.pm
package MyApp;
use Dancer2;

# another useful feature:
set auto_page => 1;

get '/' => sub { template 'index' => {...} };

# MyApp/API.pm
package MyApp::API;
use Dancer2;
set serializer => 'JSON'; # or any other serializer

get '/' => sub { +{ resources => \%resources, ... } };

# user-specific routes, for example
prefix => '/users' => sub {
    get '/view'     => sub {...};
    get '/view/:id' => sub {...};
    put '/add'      => sub {...}; # automatically deserialized params
};

...
```

Then those will be mounted together for a single app:

```
# handler: app.pl:
use MyApp;
use MyApp::API;
use Plack::Builder;

builder {
    mount '/'    => MyApp->to_app;
    mount '/api' => MyApp::API->to_app;
};
```

An example: Writing API interfaces

This example demonstrates an app that makes a request to a weather API and then displays it dynamically in a web page.

Other than Dancer2 for defining routes, we will use HTTP::Tiny to make the weather API request, JSON to decode it from JSON format, and finally File::Spec to provide a fully-qualified path to our

template engine.

```
use JSON;
use Dancer2;
use HTTP::Tiny;
use File::Spec;
```

Configuration

We use the Template::Toolkit template system for this app. Dancer searches for our templates in our views directory, which defaults to *views* directory in our current directory. Since we want to put our template in our current directory, we will configure that. However, *Template::Toolkit* does not want us to provide a relative path without configuring it to allow it. This is a security issue. So, we're using File::Spec to create a full path to where we are.

We also unset the default layout, so Dancer won't try to wrap our template with another one. This is a feature in Dancer to allow you to wrap your templates with a layout when your templating system doesn't support it. Since we're not using a layout here, we don't need it.

```
set template => 'template_toolkit';       # set template engine
set layout   => undef;                    # disable layout
set views    => File::Spec->rel2abs('.'); # full path to views
```

Now, we define our URL:

```
my $url = 'http://api.openweathermap.org/data/2.5/weather?id=5110629&units=i
```

Route

We will define a main route which, upon a request, will fetch the information from the weather API, decode it, and then display it to the user.

Route definition:

```
get '/' => sub {
    ...
};
```

Editing the stub of route dispatching code, we start by making the request and decoding it:

```
# fetch data
my $res = HTTP::Tiny->new->get($url);

# decode request
my $data = decode_json $res->{'content'};
```

The data is not just a flat hash. It's a deep structure. In this example, we will filter it for only the simple keys in the retrieved data:

```
my $metrics = { map +(
    ref $data->{$_} ? () : ( $_ => $data->{$_} )
), keys %{$data} };
```

All that is left now is to render it:

```
template index => { metrics => $metrics };
```

NON-STANDARD STEPS

Turning off warnings

The `warnings` pragma is already used when one loads Dancer2. However, if you *really* do not want the `warnings` pragma (for example, due to an undesired warning about use of undef values), add a `no warnings` pragma to the appropriate block in your module or psgi file.

AUTHOR

Dancer Core Developers

COPYRIGHT AND LICENSE