What is Dancer2?

Dancer2 is a "micro" web framework which is modeled after a Ruby framework called *Sinatra* that constructs web applications by building a list of HTTP verbs, URLs (called routes) and methods to handle that type of traffic to that specific URL.

```
use Dancer2;

get '/' => sub {
  return 'Hello World!';
};

start;
```

This example shows a single HTTP verb "GET" followed by the root URL "/" and an anonymous subroutine which returns the string `"Hello World!"` If you were to run this example, it would display "Hello World!" when you point your browser at *http://localhost:3000*.

How about a little more involved example?

That's the reason I wrote this tutorial. While I was investigating some Python web frameworks like *Flask* or *Bottle* I enjoyed the way they explained step by step how to build an example application which was a little more involved than a trivial example.

Using the *Flaskr* sample application as my inspiration (OK, shamelessly plagiarised) I translated that application to the Dancer2 framework so I could better understand how Dancer2 worked. (I'm learning it too!)

So "Dancr" was born.

Dancr is a simple "micro" blog which uses the *SQLite* database engine for simplicity's sake. (You'll need to install sqlite if you don't have it installed already.)

Required perl modules

Obviously you need *Dancer2*. You also need the *Template Toolkit*, *File::Slurper*, and *DBD::SQLite*. These all can be installed using your CPAN client, as in:

```
cpan Dancer2 Template File::Slurper DBD::SQLite
```

The database

We're not going to spend a lot of time on the database, as it's not really the point of this particular tutorial. Open your favorite *text editor* and create a schema definition called 'schema.sql' with the following content:

```
create table if not exists entries (
  id integer primary key autoincrement,
  title string not null,
  text string not null
);
```

Here we have a single table with three columns: id, title, and text. The 'id' field is the primary key and will automatically get an ID assigned by the database engine when a row is inserted.

We want our application to initialize the database automatically for us when we start it, so next, create a file called 'dancr.pl'. (The entire file is listed below, so don't worry about copying each of these fragments into 'dancr.pl' as you read through this document.) We're going to put the following subroutines in that file:

```
sub connect_db {
```

```
    my $dbh = DBI->connect("dbi:SQLite:dbname=".setting('database')) or
       die $DBI::errstr;

    return $dbh;
}

sub init_db {
    my $db = connect_db();
    my $schema = read_text('./schema.sql');
    $db->do($schema) or die $db->errstr;
}
```

Nothing too fancy in here, I hope. Standard DBI except for the `setting('database')` thing - more on that in a bit. For now, just assume that the expression evaluates to the location of the database file.

(Note that you may want to look at the *Dancer2::Plugin::Database* module for an easy way to configure and manage database connections for your Dancer2 apps, but the above will suffice for this tutorial.)

Our first route handler

Let's tackle our first route handler now, the one for the root URL '/'. This is what it looks like:

```
get '/' => sub {
    my $db = connect_db();
    my $sql = 'select id, title, text from entries order by id desc';
    my $sth = $db->prepare($sql) or die $db->errstr;
    $sth->execute or die $sth->errstr;
    template 'show_entries.tt', {
       'msg' => get_flash(),
       'add_entry_url' => uri_for('/add'),
       'entries' => $sth->fetchall_hashref('id'),
    };
};
```

As you can see, the handler is created by specifying the HTTP verb 'get', the '/' URL to match, and finally, a subroutine to do something once those conditions have been satisfied. Something you might not notice right away is the semicolon at the end of the route handler. Since the subroutine is actually a coderef, it requires a semicolon.

Let's take a closer look at the subroutine. The first few lines are standard DBI. The only new concept as part of Dancer2 is that `template` directive at the end of the handler. That tells Dancer2 to process the output through one of its templating engines. In this case, we're using *Template Toolkit* which offers a lot more flexibility than the simple default Dancer2 template engine.

Templates all go into the `views/` directory. Optionally, you can create a "layout" template which provides a consistent look and feel for all of your views. We'll construct our own layout template cleverly named *main.tt* a little later in this tutorial.

What's going on with the hashref as the second argument to the template directive? Those are all of the parameters we want to pass into our template. We have a `msg` field which displays a message to the user when an event happens like a new entry is posted, or the user logs in or out. It's called a "flash" message because we only want to display it one time, not every time the / URL is rendered.

The `uri_for` directive tells Dancer2 to provide a URI for that specific route, in this case, it is the route to post a new entry into the database. You might ask why we don't simply hardcode the `/add` URI in our application or templates. The best reason **not** to do that is because it removes a layer of flexibility as to where to "mount" the web application. Although the application is coded to use the root

URL `/` it might be better in the future to locate it under its own URL route (maybe `/dancr`?) - at that point we'd have to go through our application and the templates and update the URLs and hope we didn't miss any of them. By using the `uri_for` Dancer2 method, we can easily load the application wherever we like and not have to modify the application at all.

Finally, the `entries` field contains a hashref with the results from our database query. Those results will be rendered in the template itself, so we just pass them in.

So what does the *show_entries.tt* template look like? This:

```
  [% IF session.logged_in %]
    <form action="[% add_entry_url %]" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  [% END %]
  <ul class=entries>
  [% IF entries.size %]
    [% FOREACH id IN entries.keys.nsort %]
      <li><h2>[% entries.$id.title | html %]</h2>[% entries.$id.text | html
 %]
    [% END %]
  [% ELSE %]
    <li><em>Unbelievable. No entries here so far</em>
  [% END %]
  </ul>
```

Again, since this isn't a tutorial specifically about Template Toolkit, I'm going to gloss over the syntax here and just point out the section which starts with `<ul class=entries>` - this is the section where the database query results are displayed. You can also see at the very top some discussion about a session - more on that soon.

The only other Template Toolkit related thing that has to be mentioned here is the `| html` in `[% entries.$id.title | html %]`. That's *a filter* to convert characters like < and > to `<` and `>`. This way they will be displayed by the browser as content on the page rather than just included. If we did not do this, the browser might interpret content as part of the page, and a malicious user could smuggle in all kinds of bad code that would then run in another user's browser. This is called *Cross Site Scripting* or XSS and you should make sure to avoid it by always filtering data that came in from the web when you display it in a template.

Other HTTP verbs

There are 8 defined HTTP verbs defined in *RFC 2616*: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT. Of these, the majority of web applications focus on the verbs which closely map to the CRUD (Create, Retrieve, Update, Delete) operations most database-driven applications need to implement.

In addition, the `PATCH` verb was defined in *RFC5789*, and is intended as a "partial PUT" - sending just the changes required to the entity in question. How this would be handled is down to your app, it will vary depending on the type of entity in question and the serialization in use.

Dancer2 currently supports GET, PUT/PATCH, POST, DELETE, OPTIONS which map to Retrieve, Update, Create, Delete respectively. Let's take a look now at the `/add` route handler which handles a POST operation.

```
post '/add' => sub {
   if ( not session('logged_in') ) {
      send_error("Not logged in", 401);
   }

   my $db = connect_db();
   my $sql = 'insert into entries (title, text) values (?, ?)';
   my $sth = $db->prepare($sql) or die $db->errstr;
   $sth->execute(
       body_parameters->get('title'),
       body_parameters->get('text')
   ) or die $sth->errstr;

   set_flash('New entry posted!');
   redirect '/';
};
```

As before, the HTTP verb begins the handler, followed by the route, and a subroutine to do something - in this case, it will insert a new entry into the database.

The first check in the subroutine is to make sure the user sending the data is logged in. If not, the application returns an error and stops processing. Otherwise, we have standard DBI stuff. Let me insert (heh, heh) a blatant plug here for always, always using parameterized INSERTs in your application SQL statements. It's the only way to be sure your application won't be vulnerable to SQL injection. (See *http://www.bobby-tables.com* for correct INSERT examples in multiple languages.) Here we're using the `body_parameters` convenience method to pull in the parameters in the current HTTP request. (You can see the 'title' and 'text' form parameters in the *show_entries.tt* template above.) Those values are inserted into the database, then we set a flash message for the user and redirect her back to the root URL.

It's worth mentioning that the "flash message" is not part of Dancer2, but a part of this specific application. We need to implement it ourself.

```
sub set_flash {
    my $message = shift;

    session flash => $message;
}

sub get_flash {
    my $msg = session('flash');
    session->delete('flash');

    return $msg;
}
```

We need a way to save our temporary message, and a way to get it back out. Since it is a temporary message that should only be shown on the page immediately following the one where the value was set, we need to delete it once it was read.

Sessions and logins

A good way to implement this "flash message" mechanic is to write the message to the user's session. The session stores data for a specific user for the whole time she uses the application. It persists over all requests this user makes. You can read more about how to use the session in *our manual*.

Dancer2 comes with a simple in-memory session manager out of the box. It supports a bunch of other session engines including YAML, memcached, browser cookies and others. For this application we're going to stick with the in-memory model which works great for development and tutorials, but won't persist across server restarts or scale very well in "real world" production scenarios.

Configuration options

To use sessions in our application, we have to tell Dancer2 to activate the session handler and initialize a session manager. To do that, we add some configuration directives toward the top of our 'dancr.pl' file. But there are more options than just the session engine we want to set.

```
  set 'database'     => File::Spec->catfile(File::Spec->tmpdir(),
'dancr.db');
  set 'session'      => 'Simple';
  set 'template'     => 'template_toolkit';
  set 'logger'       => 'console';
  set 'log'          => 'debug';
  set 'show_errors'  => 1;
  set 'startup_info' => 1;
  set 'warnings'     => 1;
```

Hopefully these are fairly self-explanatory. We want the Simple session engine, the Template Toolkit template engine, logging enabled (at the 'debug' level with output to the console instead of a file), we want to show errors to the web browser, log access attempts and log Dancer2 warnings (instead of silently ignoring them).

In a more sophisticated application you would want to put these configuration options into a configuration file, but for this tutorial, we're going to keep it simple. Dancer2 also supports the notion of application environments, meaning you can create a configuration file for your development instance, and another config file for the production environment (with things like debugging and showing errors disabled perhaps). Dancer2 also doesn't impose any limits on what parameters you can set using the `set` syntax. For this application we're going to embed our single username and password into the application itself:

```
  set 'username' => 'admin';
  set 'password' => 'password';
```

Hopefully no one will ever guess our clever password! Obviously, you will want a more sophisticated user authentication scheme in any sort of non-tutorial application but this is good enough for our purposes.

Logging in

Now that Dancr is configured to handle sessions, let's take a look at the URL handler for the `/login` route.

```
  any ['get', 'post'] => '/login' => sub {
     my $err;

     if ( request->method() eq "POST" ) {
       # process form input
       if ( body_parameters->get('username') ne setting('username') ) {
         $err = "Invalid username";
       }
       elsif ( body_parameters->get('password') ne setting('password') ) {
         $err = "Invalid password";
       }
       else {
         session 'logged_in' => true;
```

```
        set_flash('You are logged in.');
        return redirect '/';
      }
  }

  # display login form
  template 'login.tt', {
    'err' => $err,
  };
};
```

This is the first handler which accepts two different verb types, a GET for a human browsing to the URL and a POST for the browser to submit the user's input to the web application. Since we're handling two different verbs, we check to see what verb is in the request. If it's **not** a POST, we drop down to the `template` directive and display the *login.tt* template:

```
<h2>Login</h2>
[% IF err %]<p class=error><strong>Error:</strong> [% err %][% END %]
<form action="[% login_url %]" method=post>
  <dl>
    <dt>Username:
    <dd><input type=text name=username>
    <dt>Password:
    <dd><input type=password name=password>
    <dd><input type=submit value=Login>
  </dl>
</form>
```

This is even simpler than our *show_entries.tt* template - but wait - there's a `login_url` template parameter and we're only passing in the `err` parameter. Where's the missing parameter? It's being generated and sent to the template in a `before_template_render` directive - we'll come back to that in a moment or two.

So the user fills out the *login.tt* template and submits it back to the `/login` route handler. We now check the user input against our application settings and if the input is incorrect, we alert the user, otherwise the application starts a session and sets the `logged_in` session parameter to the `true()` value. Dancer2 exports both a `true()` and `false()` convenience method which we use here. After that, it's another flash message and back to the root URL handler.

Logging out

And finally, we need a way to clear our user's session with the customary logout procedure.

```
get '/logout' => sub {
   app->destroy_session;
   set_flash('You are logged out.');
   redirect '/';
};
```

`app->destroy_session;` is Dancer2's way to remove a stored session. We notify the user she is logged out and route her back to the root URL once again.

You might wonder how we can then set a value in the session in `set_flash`, because we just destroyed the session.

Destroying the session has removed the data from the persistence layer (which is the memory of our running application, because we are using the `simple` session engine). If we write to *the session* now, it will actually create a completely new session for our user. This new, empty session will have a

new *session ID*, which Dancer2 tells the user's browser about in the response. When the browser requests the root URL, it will send this new session ID to our application.

Layout and static files

We still have a missing puzzle piece or two. First, how can we use Dancer2 to serve our CSS stylesheet? Second, where are flash messages displayed? Third, what about the `before_template_render` directive?

Serving static files

In Dancer2, static files should go into the `public/` directory, but in the application itself be sure to omit the `public/` element from the path. For example, the stylesheet for Dancr lives in `dancr/public/css/style.css` but is served from *http://localhost:3000/css/style.css*.

If you wanted to build a mostly static web site you could simply write route handlers like this one:

```
get '/' => sub {
   send_file 'index.html';
};
```

where index.html would live in your `public/` directory.

`send_file` does exactly what it says: it loads a static file, then sends the contents of that file to the user.

Layouts

I mentioned near the beginning of this tutorial that it is possible to create a `layout` template. In Dancr, that layout is called `main` and it's set up by putting in a directive like this:

```
set layout => 'main';
```

near the top of your web application. This tells Dancer2's template engine that it should look for a file called *main.tt* in `dancr/views/layouts/` and insert the calls from the `template` directive into a template parameter called `content`.

For this web application, the layout template looks like this:

```
<!doctype html>
<html>
<head>
  <title>Dancr</title>
  <link rel=stylesheet type=text/css href="[% css_url %]">
</head>
<body>
  <div class=page>
  <h1>Dancr</h1>
     <div class=metanav>
     [% IF not session.logged_in %]
       <a href="[% login_url %]">log in</a>
     [% ELSE %]
       <a href="[% logout_url %]">log out</a>
     [% END %]
  </div>
  [% IF msg %]
    <div class=flash> [% msg %] </div>
  [% END %]
  [% content %]
</div>
</body>
```

```
  </html>
```

Aha! You now see where the flash message `msg` parameter gets rendered. You can also see where the content from the specific route handlers is inserted (the fourth line from the bottom in the `content` template parameter).

But what about all those other `*_url` template parameters?

Using before_template_render

Dancer2 has a way to manipulate the template parameters before they're passed to the engine for processing. It's `before_template_render`. Using this directive, you can generate and set the URIs for the `/login` and `/logout` route handlers and the URI for the stylesheet. This is handy for situations like this where there are values which are re-used consistently across all (or most) templates. This cuts down on code-duplication and makes your app easier to maintain over time since you only need to update the values in this one place instead of everywhere you render a template.

```
  hook before_template_render => sub {
     my $tokens = shift;

     $tokens->{'css_url'} = request->base . 'css/style.css';
     $tokens->{'login_url'} = uri_for('/login');
     $tokens->{'logout_url'} = uri_for('/logout');
  };
```

Here again I'm using `uri_for` instead of hardcoding the routes. This code block is executed before any of the templates are processed so that the template parameters have the appropriate values before being rendered.

Putting it all together

Here's the complete 'dancr.pl' script from start to finish.

```
 use Dancer2;
 use DBI;
 use File::Spec;
 use File::Slurper qw/ read_text /;
 use Template;

 set 'database'     => File::Spec->catfile(File::Spec->tmpdir(),
'dancr.db');
 set 'session'      => 'Simple';
 set 'template'     => 'template_toolkit';
 set 'logger'       => 'console';
 set 'log'          => 'debug';
 set 'show_errors'  => 1;
 set 'startup_info' => 1;
 set 'warnings'     => 1;
 set 'username'     => 'admin';
 set 'password'     => 'password';
 set 'layout'       => 'main';

 sub set_flash {
     my $message = shift;

     session flash => $message;
 }
```

```
sub get_flash {
    my $msg = session('flash');
    session->delete('flash');

    return $msg;
}

sub connect_db {
    my $dbh = DBI->connect("dbi:SQLite:dbname=".setting('database')) or
        die $DBI::errstr;

    return $dbh;
}

sub init_db {
    my $db = connect_db();
    my $schema = read_text('./schema.sql');
    $db->do($schema) or die $db->errstr;
}

hook before_template_render => sub {
    my $tokens = shift;

    $tokens->{'css_url'} = request->base . 'css/style.css';
    $tokens->{'login_url'} = uri_for('/login');
    $tokens->{'logout_url'} = uri_for('/logout');
};

get '/' => sub {
    my $db = connect_db();
    my $sql = 'select id, title, text from entries order by id desc';
    my $sth = $db->prepare($sql) or die $db->errstr;
    $sth->execute or die $sth->errstr;
    template 'show_entries.tt', {
        'msg' => get_flash(),
        'add_entry_url' => uri_for('/add'),
        'entries' => $sth->fetchall_hashref('id'),
    };
};

post '/add' => sub {
    if ( not session('logged_in') ) {
        send_error("Not logged in", 401);
    }

    my $db = connect_db();
    my $sql = 'insert into entries (title, text) values (?, ?)';
    my $sth = $db->prepare($sql) or die $db->errstr;
    $sth->execute(
        body_parameters->get('title'),
        body_parameters->get('text')
    ) or die $sth->errstr;

    set_flash('New entry posted!');
```

```
    redirect '/';
};

any ['get', 'post'] => '/login' => sub {
    my $err;

    if ( request->method() eq "POST" ) {
        # process form input
        if ( body_parameters->get('username') ne setting('username') ) {
            $err = "Invalid username";
        }
        elsif ( body_parameters->get('password') ne setting('password') )
{
            $err = "Invalid password";
        }
        else {
            session 'logged_in' => true;
            set_flash('You are logged in.');
            return redirect '/';
        }
   }

   # display login form
   template 'login.tt', {
       'err' => $err,
   };

};

get '/logout' => sub {
   app->destroy_session;
   set_flash('You are logged out.');
   redirect '/';
};

init_db();
start;
```

Advanced route moves

There's a lot more to route matching than shown here. For example, you can match routes with regular expressions, or you can match pieces of a route like `/hello/:name` where the `:name` piece magically turns into a named parameter in your handler for manipulation.

Next Steps

Hopefully this effort has been helpful and interesting enough to get you exploring Dancer2 on your own. Dancer2 is suitable for projects of all shapes and sizes, and you can easily build a solution tailored to your particular needs.

There is certainly a lot left to cover. We suggest you take a stroll through the *manual*for more detailed information about the topics covered in this tutorial, and for a more thorough explanation of all things Dancer2-related. The *cookbook* will show you how to handle some handy tips and tricks when working with Dancer2. Additionally, there are a lot of great *plugins* which extend and enhance the capabilities of the framework.

Happy dancing!

SEE ALSO

- *http://perldancer.org*
- *http://github.com/PerlDancer/Dancer2*
- *Dancer2::Plugins*

CSS COPYRIGHT AND LICENSE