Tutorial Overview

This tutorial has three parts. Since they build on one another, each part is meant to be gone through in sequential order.

Part I, the longest section, focuses on the basics of Dancer2 development by building a simple yet functional blog app, called `dancr`, that you can use to impress your friends and family.

In **Part II**, you'll learn about the preferred way to get web apps up and running with the `dancer2` utility by porting the script written in Part I into a new-and-improved Dancer2 app, called `Dancr2`.

Finally, in **Part III**, we give you a taste of the power of plugins that other developers have written by modifying the `Dancr2` app to use a database plugin.

This tutorial assumes you have some familiarity with Perl and that you know how to create and execute a Perl script on your computer. Some experience with web development is also helpful but not entirely necessary. This tutorial is mostly geared toward web developers but website designers can get something out of it as well since the basics of templating are covered. Plus, it might be good for a designer to have a decent idea of how Dancer2 works.

Part I: Let's Get Dancing!

Part I covers many of the basic concepts you'll need to know to lay a good foundation for your future development work with Dancer2 by building a simple micro-blogging app.

What is Dancer2?

Dancer2 is a micro-web framework, written in the Perl programming language, and is modeled after a Ruby web application framework called *Sinatra*.

When we say "micro" framework, we mean that Dancer2 aims to maximize your freedom and control by getting out of your way. "Micro" doesn't mean Dancer2 is only good for creating small apps. Instead, it means that Dancer2's primary focus is on taking care of a lot of the boring, technical details of your app for you and by creating an easy, clean routing layer on top of your app's code. It also means you have almost total control over the app's functionality and how you create and present your content. You will not confined to someone else's approach to creating a website or app.

Dancer2 can build a specialized content management system or provide a full-fledged API for querying a database over the web. Dancer2 has hundreds of plugins that you can take advantage of so you don't have to reinvent the wheel. You can add only the capabilities your app needs to keep complexity to a minimum.

As a framework, Dancer2 provides you with the tools and infrastructure you can leverage to deliver content on the web quickly, easily and securely. The tools Dancer2 provides, called "keywords," are commands that you use to build your app, access the data inside of it, and deliver it on the internet in many different formats.

Dancer2's keywords provide what is called a **Domain Specific Language** (DSL) designed specifically for the task of building apps. But don't let the technical jargon scare you off. Things will become clearer shortly in our first code example.

Getting Dancer2 installed First, we need to make sure you have Dancer2 installed. Typically, you will do that with one of the following two commands:

```
  cpan Dancer2  # requires the cpan command to be installed and configured
  cpanm Dancer2 # requires you have cpanimus installed and available on
your OS
```

If you aren't familiar with installing Perl modules on your machine, you should *read this guide*. You may also want to consult your OS's documentation or a knowledgeable expert. And, of course, your search engine of choice is always there for you, as well.

Your first Dancer2 "Hello World!" app After getting Dancer2 installed, open up your favorite text editor and copy and paste the following lines of Perl code into it and save it to a file called dancr.pl:

```
  #!/usr/bin/perl   # Don't forget to change your path to Perl as
necessary.
  use Dancer2;

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

  start;
```

If you make this script executable and run it, it will fire up a simple, standalone web server that will display "Hello World!" when you point your browser to *http://localhost:3000*. Cool!

Important note: We want to emphasize that writing a script file like this with a `start` command is **not** how you would typically begin writing a Dancer2 app. Part II of this tutorial will show you the recommended approach using the `dancer2` utility. We are using this approach in order to stay focused on the fundamentals.

Though our example app is very simple, there is a lot going on under the hood, especially with our first line of code, `use Dancer2;`. We won't go into the gory technical details of how that works. For now, it's enough for you to know that the Dancer2 module infuses your script with the ability to use Dancer2 keywords. Getting comfortable with Dancer2 keywords is probably the most important step you can take as a budding Dancer2 developer and this tutorial will do its best to help foster your understanding of them.

The next line of code in our example (which spans three lines to make it more readable) is the **route handler**. Let's examine this line closely, because route handlers are at the core of how to build an app with Dancer2.

The syntax of a Dancer2 `route handler` has three parts:

* an **http method** or **http verb**; in this example, we use the `get` keyword to tell Dancer2 that this route should apply to GET http requests. `get` is the first of many keywords that Dancer2 provides that we will cover in this tutorial. Those familiar with web development will know that a GET request is what we use to fetch information from a website.

* the **route pattern**; this is the bit of code that appears immediately after our `get` keyword. In this example it is a forward slash (`/`), wrapped in single quotes, and it represents the pattern we wish to match against the URL that the browser, or client, has requested. Web developers will immediately recognize that the forward slash symbolizes the root directory of our website. Experienced Perl programmers will pick up on the fact that the route pattern is nothing more than an argument for our `get` keyword.

* the **route action**; this is the subroutine that returns our data. More precisely, it is a subroutine reference. The route action in our example returns a simple string, `Hello World!`. Like the route pattern, the route action is nothing more than an argument to our `get` keyword.

> Note that convention has us use the fat arrow (=>) operator between the route pattern and the action to make our code more readable. But we could just as well have used a regular old comma to separate these argument to our `get` method. Gotta love Perl for its flexibility.

Putting our route pattern in the example above into plain English, we tell our app, "If the root directory is requested with the GET http method, send the string 'Hello Word!' back in our response." Of course, since this is a web app, we also have to send back headers with our response. This is quitely taken care of for us by Dancer2 so we don't have to think about it.

The syntax for route handlers might seem a bit foreign for newer Perl developers. But rest assured there is nothing magical about it and it is all just plain old Perl under the hood. If you keep in mind that

the keyword is a subroutine (or more precisely, a method) and that the pattern and action are arguments to the keyword, you'll pick it up in no time. Thinking of these keywords as "built-ins" to the Dancer2 framework might also eliminate any initial confusion about them.

The most important takeaway here is that we build our app by adding route handlers which are nothing more than a collection of, HTTP verbs, URL patterns, and actions.

How about a little more involved example?

While investigating some Python web frameworks like *Flask* or *Bottle*, I enjoyed the step-by-step explanation for building an sample application which was a little more involved than a trivial example. This tutorial is modeled after them.

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 using the *SQLite* database engine for simplicity's sake. You'll need to install sqlite on your machine if you don't have it installed already. Consult your OS documentation for getting SQLite installed on your machine.

Required perl modules

In addition to the *Dancer2* module, you'll also need the *Template Toolkit*, *File::Slurper*, and *DBD::SQLite* modules. These can all be installed using your CPAN client with the following command:

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

Add the following lines to your `dancr.pl` file to import the modules you just installed:

```
use DBI;
use File::Slurper qw / read_text /;
use File::Spec;  # part of Perl core, used to provide a path to our DB
use Template;
```

The database code

We're not going to spend a lot of time on the database, as it's a bit beyond the scope of this tutorial. Try not to dwell on this section too much if you don't understand all of it.

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, let's edit the 'dancr.pl' file we created earlier and give it the ability to talk to our database with the following subroutines: (Or, if you prefer, you can copy and paste the finished dancr.pl scriptâ€“found near the end of Part I in this tutorialâ€“into the file all at once and then just follow along with the tutorial.)

```
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. It's standard DBI except for the `setting('database')` thingâ€”more on that in a bit. For now, just assume that the expression evaluates to the location of the database file. We also need a line in our script to call the `init_db` subroutine. Add the following line just above the `start;` line at the end of the file:

```
init_db();
```

In Part III of the tutorial, we will show you how to use the *Dancer2::Plugin::Database* module for an easier way to configure and manage database connections for your Dancer2 apps.

Our first route handler

Ok, let's get back to the business of learning Dancer2 by providing a more useful route handler for the root URL. Replace the route handler in our simple example above with this one:

```
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'),
  };
};
```

Our new route handler is the same as the one in our first example except that our route action does a lot more work.

Something you might not have noticed right away is the semicolon at the end of the route handler. This might confuse newer Perl coders and is a source of bugs for more experienced ones who forget to add it. We need the semicolon there because we are creating a reference to a subroutine and because that's just what the Perl compiler demands and we must obey if we want our code to run.

Alright, let's take a closer look at this route's action. The first few lines are standard DBI. The really important bit related to Dancer2 is the `template` keyword at the end of the action. That tells Dancer2 to process the output through one of its templating engines. There are many template engines available for use with Dancer2. In this tutorial, we're using *Template Toolkit* which offers a lot more flexibility than the simple default Dancer2 template engine.

The templates our app will use are placed in the `views/` directory which are located in the same directory as our dancr.pl script. 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.

So 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 templates. We see a `msg` field which displays a message to the user when an event happens like when a new entry gets posted, or the user logs in or out. We call it a "flash" message not because it blinks but because we only want to display it one time, not

every time the `/` URL is rendered. We see that `msg` calls a `get_flash` subroutine. `get_flash` is not a Dancer2 function and is our own invention. So we need to manually add this subroutine along with its associated variable by adding the following code near the top of our script:

```
my $flash;

sub get_flash {
  my $msg = $flash;
  $flash = "";

  return $msg;
}
```

This will become more clear when we discuss the flash messages in more detail later.

Moving on, 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>
```

Go ahead and create a `views/` directory in the same directory as the script and add this file to it.

Again, since this isn't a tutorial about Template Toolkit, we'll gloss over the syntax here and just point out the section which starts with `<ul class=entries>`â€“this is the section where the database query results are displayed. You can also see at the very top some discussion about a sessionâ€“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's keywords 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 somethingâ€“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.

Before moving on to our next topic, we need to add a subroutine to our script for handling the call to `set_flash` just as we did with `get_flash`:

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

```
    $flash = $message;
}
```

Logins and sessions

Since the HTTP protocol is stateless, web applications need a session manager to determine if a user has already been authenticated. Dancer2 comes with a simple in-memory session manager out of the box. It supports many other session engines including YAML, memcached, browser cookies and others. We'll use the simple in-memory model which works great for development and tutorials. But because simple sessions don't persist across server restarts or scale very well, they aren't suitable for "real world" web applications.

Configuration options

To use the simple session manager in our application, we add a configuration directives toward the top of our 'dancr.pl' file. While doing that, we'll show you some other configuration settings that we want as well:

```
  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;
```

Most of these are fairly self-explanatory. First we tell Dancer to find our database file using `File::Spec` to generate a path that works across different operating systems. We then tell it to use the Simple session engine; use the Template Toolkit template engine; enable logging (at the 'debug' level with output to the console instead of a file); show errors to the web browser; and log access attempts and log Dancer2 warnings (as opposed to silently ignoring them).

Dancer2 doesn't impose any limits on creating parameters with the `set` syntax and we can set up our own. For this application we're going to embed our single username and password into the application itself:

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

Obviously, you'll want a far more secure and sophisticated authentication scheme for your real-world application but this is good enough for our purposes.

In Part II, we'll use Dancer2's configuration files to manage these configuration options and set up different environments for the app with configuration files.

Logging in

Now that dancr can 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 handler 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, or if a login attempt failed, we drop down to the `template` directive and display the *login.tt* template along with errors, if any:

```
<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* templateâ€“but waitâ€“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` directiveâ€“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.

Layout and static files

We still have a missing puzzle piece or two. First, how do we add css to our page layout? Second, where are flash messages displayed? Third, what about the `before_template_render` directive mentioned earlier?

Serving static files

In Dancer2, static files should go into the `public/` directory. However, in the application, omit the `public/` element from the path. For example, the stylesheet for dancr will live 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 and serves the contents of the file out to the user.

Let's go ahead and create our style sheet. In the same directory as your dancr.pl script, issue the following commands:

```
mkdir public && mkdir public/css && touch public/css/style.css
```

Next, add the following css to the `public/css/style.css` file you just created:

```
body            { font-family: sans-serif; background: #eee; }
a, h1, h2       { color: #377ba8; }
h1, h2          { font-family: 'Georgia', serif; margin: 0; }
h1              { border-bottom: 2px solid #eee; }
h2              { font-size: 1.2em; }

.page           { margin: 2em auto; width: 35em; border: 5px solid #ccc;
                  padding: 0.8em; background: white; }
.entries        { list-style: none; margin: 0; padding: 0; }
.entries li     { margin: 0.8em 1.2em; }
.entries li h2  { margin-left: -1em; }
.add-entry      { font-size: 0.9em; border-bottom: 1px solid #ccc; }
.add-entry dl   { font-weight: bold; }
.metanav        { text-align: right; font-size: 0.8em; padding: 0.3em;
                  margin-bottom: 1em; background: #fafafa; }
.flash          { background: #cee5F5; padding: 0.5em;
                  border: 1px solid #aacbe2; }
.error          { background: #f0d6d6; padding: 0.5em; }
```

Layouts

I mentioned earlier in the 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 the template engine that to look for a file called *main.tt* in `views/layouts/` and to insert the calls from the `template` directive into a template parameter called `content`.

Here is the simple layout file we will use for this web application. Go ahead and add the *main.tt* file to the `views/layouts/` directory.

```
<!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 displayed. 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 keyword, 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.

One Last Thing

There is one other minor detail and that is our app lacks a `favicon.ico` file. Without it, our log will report annoying file not found errors it. To keep our log quiet, run the following command from the dancr script's directory:

```
 touch public/favicon.ico
```

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';

my $flash;

sub set_flash {
    my $message = shift;

    $flash = $message;
}

sub get_flash {

    my $msg = $flash;
    $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.

You can explore this and other advanced concepts by reading the *Dancer2::Manual*.

Part II: Taking Advantage of the dancer2 Utility to Set Up New Apps

In Part I, we took a simple Perl script and transformed it into a basic web app to teach you basic Dancer2 concepts. While starting with a simple script like this helped make it easier to teach these concepts, it did not demonstrate how a typical app is built by a Dancer2 developer. So let's show you how things really get done.

Creating a new app

Now that you have a clearer idea of what goes into building an app with Dancer2, it's time to cha-cha with the `dancer2` utility which will save you a lot of time and effort by setting up directories, files, and default configuration settings for you.

The `dancer2` CLI utility was installed on your machine when you installed the Dancer2 distribution. Hop over to the command line into a directory you have permission to write to and issue the following command:

```
dancer2 -a Dancr2
```

That command should output something like the following to the console:

```
    + Dancr2
    + Dancr2/config.yml
    + Dancr2/Makefile.PL
    + Dancr2/MANIFEST.SKIP
    + Dancr2/.dancer
    + Dancr2/cpanfile
    + Dancr2/bin
    + Dancr2/bin/app.psgi
    + Dancr2/environments
    + Dancr2/environments/development.yml
    + Dancr2/environments/production.yml
    + Dancr2/lib
    + Dancr2/lib/Dancr2.pm
    + Dancr2/public
```

```
+ Dancr2/public/favicon.ico
+ Dancr2/public/500.html
+ Dancr2/public/dispatch.cgi
+ Dancr2/public/404.html
+ Dancr2/public/dispatch.fcgi
+ Dancr2/public/css
+ Dancr2/public/css/error.css
+ Dancr2/public/css/style.css
+ Dancr2/public/images
+ Dancr2/public/images/perldancer.jpg
+ Dancr2/public/images/perldancer-bg.jpg
+ Dancr2/public/javascripts
+ Dancr2/public/javascripts/jquery.js
+ Dancr2/t
+ Dancr2/t/001_base.t
+ Dancr2/t/002_index_route.t
+ Dancr2/views
+ Dancr2/views/index.tt
+ Dancr2/views/layouts
+ Dancr2/views/layouts/main.tt
```

What you just did was create a fully functional app in Dancer2 with just one command! The new app, named "Dancr2," won't do anything particularly useful until you add your own routes to it, but it does take care of many of the tedious tasks of setting up an app for you.

The files and folders that were generated and that you see listed above provide a convenient scaffolding, or **skeleton**, upon which you can build your app. The default skeleton provides you with basic error pages, css, javascript, graphics, tests, templates and other files which you are free to modify and customize to your liking.

If you don't like the default skeleton, the `dancer2` command allows you to generate your own custom skeletons. Consult *"BOOTSTRAPPING_A_NEW_APP" in Dancer2::Manual* for further details on this and other capabilities of the `dancer2`) utility.

Getting the new app up and running with Plack

In Part I, we used the `start` command in our script to launch a server to serve our app. Things are a little different when using `dancer2`, however. You'll notice that the `dancer2` utility created a `bin/` directory with a file in it called `app.psgi`. This is the file we use to get our app up and running.

Let's see how to to do that by first changing into the Dancr2 directory and then starting the server using the `plackup` command:

```
cd Dancr2;
plackup -p 5000 bin/app.psgi
```

If all went well, you'll be able to see the Dancr2 home page by visiting:

```
http://localhost:5000
```

The web page you see there gives you some very basic advice for tuning and modifying your app and where you can go for more information to learn about developing apps with Dancer2 (like this handy tutorial!).

Our Dancr2 app is served on a simple web server provided by Plack. Plack is PSGI compliant software, hence the `psgi` extension for our file in the `bin/` directory. Plack and PSGI are beyond the scope of this tutorial but you can learn more by visiting the *Plack website*.

For now, all you need to know is that if you are deploying an app for use by just yourself or a handful

of people on a local network, Plack alone may do the trick. More typically, you would use Plack in conjunction with other server software to make your app much more robust. But in the early stages of your app's development, a simple Plack server is more than likely all you need.

To learn more about the different ways for deploying your app, see the *Dancer2 Deployment Manual*

Porting dancr.pl over to the new Dancr2 app

With our new Dancr2 app up and running, it's time to port our dancr.pl script created in Part I into it.

The lib/ directory The lib/ directory in our Dancr2 app is where our app.psgi file will expect our code to live. So let's take a peek at the file generated for us in there:

```
cat lib/Dancr2.pm
```

You'll see something like the following bit of code which provides a single route to our app's home page and loads the index template:

```
package Dancr2;
use Dancer2;

our $VERSION = '0.1';

get '/' => sub {
    template 'index' => { 'title' => 'Dancr2' };
};

true;
```

The first thing you'll notice is that instead of a script, we are using a module, `Dancr2` to package our code. Modules make it easer to pull off many powerful tricks like packaging our app across several discrete modules. We'll let the *manual* explain this more advanced technique.

Updating the Dancr2 module

Now that we know where to put our code, let's update the `Dancr2.pm` module with our original `dancr.pl` code. Remove the existing sample route in `Dancr2.pm` and replace it with the code from our `dancr.pl` file. You'll have to make a couple of adjustments to the `dancr.pl` code like removing the `use Dancer2;` line since it's already provided by our module. You'll also want to be sure to remove the `start;` line as well from the end of the file.

When you're done, `Dancr2.pm` should look something close to this:

```
package Dancr2;
use Dancer2;

our $VERSION = '0.1';

# Our original dancr.pl code with some minor tweaks
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'      => 'YAML';
...
```

```
  <snip> # The rest of the stuff </snip>

  ...

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

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

  init_db();
```

To avoid getting an error in the `init_db`) subroutine when it tries to load our schema file, copy over the `schema.db` file to the root directory of the Dancr2 app:

```
  cp /path/to/dancr.pl/schema.db /path/to/Dancr2;
```

Ok, now that we've got the core app code moved over, let's move the assets from dancr.pl into the appropriate app directories.

The public/ directory

As mentioned in Part I, our static assets go into our `public/` directory. If you followed along with the tutorial in Part I, you should have a `public/` directory with a `public/css` subdirectory and a file called `style.css` within that.

Dancer2 has conveniently generated the `public/css` directory for us which has a default css file. Let's copy the style sheet from our original app so our new app can use it:

```
  # Note: This command overwrites the default style sheet. Move it or copy
it if
  # you wish to preserve it.

  cp /path/to/dancr.pl/public/css/style.css /path/to/Dancr2/public/css;
```

The views directory

Along with our `public/` directory, Dancer has also provided a `views/` directory, which as we covered, serves as the a home for our templates. Let's get those copied over:

```
  # NOTE: This command will overwrite the default main.tt tempalte file.
Move it
  # or copy it if you wish to preserve it.

  cp -r /path/to/dancr.pl/views/* /path/to/Dancr2/views;
```

Does it work?

If you followed the instructions here closely, your Dancr2 app should be working. Shut down any running Plack servers and then issue the same plackup command to see if it runs:

```
  cd /path/to/Dancr2
  plackup -p 5000 bin/app.psgi
```

If you see any errors, get them resolved until the app loads.

Configuring Your App

In Part I, you configured your app with a series of `set` statements near the top of your file. Now we will show you a better way to configure your app using Dancer2's configuration files.

The default skeleton provides your app with three different configuration files. The first two files we'll discuss, found in the `environments/` folder of your app, are `development.yml` and `production.yml`. As you can probably guess, the `development.yml` file has settings intended to be used while developing the app. The `production.yml` file has settings more appropriate for running your app when used by others. The third configuration file is found in the root directory of your app and is named `config.yml`. This file has the settings that are common to all environments but that can be overridden by the environment configuration files. You can still override any configuration file settings in your app's modules using the `set` command.

We will take a look at the `development.yml` file first. Open that file in your text editor and take a look inside. It has a bunch of helpful comments and the following five settings sprinkled throughout:

```
logger: "console"
log: "core"
warnings: 1
show_errors: 1
startup_info: 1
```

The first four settings duplicate many of the settings in our new Dancr2 app. So in the spirit of DRY (don't repeat yourself), edit your Dancr2 module and delete the four lines that correspond to these four settings.

Then, in the configuration file, be sure to change the value for the `log` setting from "core" to "debug" so it matches the value we had in our module.

We will leave it up to you what you want to do with the fifth setting, `startup_info`. You can read about that setting, along with all the other settings, in the *configuration manual*.

Finally, let's add a new setting to the configuration file for `session` with the following line:

```
session: "Simple"
```

Then delete the corresponding setting from your Dancr2 module.

Alright, our Dancr2 app is a little leaner and meaner. Now open the main `config.yml` file and look for the settings in there that are also duplicated in our app's module. There are two:

```
layout: "main"
template: "simple"
```

Leave `layout` as is but change the template setting to "template_toolkit". Then edit your Dancr2 module file and delete these two settings.

Finally, add the following configuration settings to the .yml file:

```
username: "admin"
password: "password"
```

Then you delete these two settings from the Dancr2 module, as well.

So, if you have been following along, you now have only the following `set` command in your Dancr2 module, related to the database configuration:

```
set 'database'     => File::Spec->catfile(File::Spec->tmpdir(),
```

```
'dancr.db');
```

We will get rid of this setting in Part III of the tutorial. All the rest of the settings have been transferred to our configuration files. Nice!

We still have a little more cleanup we can do. Now that Dancer2 knows we are using Template::Toolkit, we can delete the `use Template;` line from our module.

Now start the app with the `plackup` command and check to see that everything works. By default, Dancer2 will load the development environment configuration. When it comes time to put your app into production, you can load the `production.yml` file configuration with plackup's `--env` switch like so:

```
plackup -p 5000 --env production bin/app.psgi
```

Keep on Dancing!

This concludes Part II of our tutorial where we showed you how to take advantage of the `dancer2` utility to set up a app skeleton to make it really easy to get started developing your own apps.

Part III will refine our app a little further by showing you how to use plugins so you can capitalize on all the great work contributed by other talented Dancer2 developers.

Part III: Plugins, Your Many Dancing Partners

Dancer2 takes advantage of the open source software revolution by making it exceedingly easy to use plugins that you can mix into your app to give it new functionality. In Part III of this tutorial, we will update our new Dancr2 app to use the *Dancer2::Plugin::Database* to give you enough skills to go out and explore other plugins on your own.

Installing plugins

Like Dancer2 itself, Dancer2 plugins can be found on the CPAN. Use your favorite method for downloading and installing the *Dancer2::Plugin::Database* module on your machine. We recommend using `cpanminus` like so:

```
cpanm Dancer2::Plugin::Database
```

Using plugins

Using a plugin couldn't be easier. Simply add the following line to your Dancr2 module below the `use Dancer2;` line in your module:

```
use Dancer2::Plugin::Database;
```

Configuring plugins

Plugins can be configured with the YAML configuration files mentioned in Part II of this tutorial. Let's edit the `development.yml` file and add our database configuration there. Below the last line in that file, add the following lines, being careful to keep the indentation as you see it here:

```
  plugins:                ; all plugin configuration settings go in this
section
    Database:             ; the name of our plugin
      driver: "SQLite"    ; driver we want to use
      database: "dancr.db" ; where the database will go in our app
                          ; run a query when connecting to the datbase:
      on_connect_do: [ "create table if not exists entries (id integer
primary key autoincrement, title string not null, text string not null)" ]
```

Here, we direct our database plugin to use the "SQLite" driver and to place the database in the root directory of our Dancr2. The `on_connect_db` setting tells the plugin to run an SQL query when it

connects with the database to create a table for us if it doesn't already exist.

Modifying our database code in the Dancr2 module

Now it's time to modify our Dancr2 module so it will use the plugin to query the database instead of our own code. There are a few things to do. First, we will delete the code we no longer need.

Since the configuration file tells the plugin where our database is, we can delete this line:

```
  set 'database'     => File::Spec->catfile(File::Spec->tmpdir(),
'dancr.db');
```

And since the database plugin will create our database connection and initialize our database for us, we can scrap the following code from our module:

```
  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;
  }

  init_db(); # Found at the bottom of our file
```

Let's now take advantage of a hook the plugin provides for handling database errors:

```
  hook 'database_error' => sub {
    my $error = shift;
    die $error;
  };
```

We'll also make a few adjustments to the bits of code that make the database queries. In our `get '/'` route, change all instances of `$db` with `database` and remove all the `die` calls since we now have a hook to handle errors. When you are done, your route should look something like this:

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

Make the same changes to the `post '/add'` route to transform it into this:

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

```
    my $sql = 'insert into entries (title, text) values (?, ?)';
    my $sth = database->prepare($sql);
    $sth->execute(
      body_parameters->get('title'),
      body_parameters->get('text')
    );

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

Our last step is to get rid of the following lines which we no longer need, thanks to our plugin:

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

That's it! Now start your app with `plackup` to make sure you don't get any errors and then point your browser to test the app to make sure it works as expected. If it doesn't, double and triple check your configuration settings and your module's code which should now look like this:

```
package Dancr2;
use Dancer2;
use Dancer2::Plugin::Database;

our $VERSION = '0.1';

my $flash;

sub set_flash {
    my $message = shift;
    $flash = $message;
}

sub get_flash {
    my $msg = $flash;
    $flash = "";
    return $msg;
}

hook 'database_error' => sub {
    my $error = shift;
    die $error;
};

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 $sql = 'select id, title, text from entries order by id desc';
  my $sth = database->prepare($sql);
  $sth->execute;
  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 $sql = 'insert into entries (title, text) values (?, ?)';
  my $sth = database->prepare($sql);
  $sth->execute(
      body_parameters->get('title'),
      body_parameters->get('text')
  );

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

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

  if ( request->method() eq "POST" ) {
    # process form input
    if ( params->{'username'} ne setting('username') ) {
        $err = "Invalid username";
    }
    elsif ( params->{'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 '/';
};

true;
```

Next steps Congrats! You are now using the database plugin like a boss. The database plugin does a lot more than what we showed you here. We'll leave it up to you to consult the Dancer2::Plugin::Database|documentation to unlock its full potential.

There are many more plugins for you to explore. You now know enough to install and experiment with them. Some of the more popular and useful plugins are listed at *Dancer2::Plugins*. You can also search CPAN with "Dancer2::Plugin" for a more comprehensive listing.

If you are feeling really inspired, you can learn how to extend Dancer2 with your own plugins by reading *Dancer2::Plugin*.

Happy dancing!

I hope these tutorials have been helpful and interesting enough to get you exploring Dancer2 on your own. The framework is still under development but it's definitely mature enough to use in a production project.

Happy dancing!

One more thing: Test! Before we go, we want to mention that Dancer2 makes it very easy to run automated tests on your app to help you find bugs. If you are new to testing, we encourage you to start learning how. Your future self will thank you. The effort you put into creating tests for your app will save you many hours of frustration in the long run. Unfortunately, until we get Part IV of this tutorial written, you'll have to consult the Dancer2 testing documentation|Dancer2::Manual/TESTING for more details on how to test your app.

Enjoy!

SEE ALSO

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

CSS COPYRIGHT AND LICENSE

The CSS stylesheet is copied verbatim from the Flaskr example application and is subject to their license: