AngularJS & Dancer

for Modern Web Development

Otherwise titled:

How I tried to get my app ready for Dancer, instead of just using Dancer, and how that worked out for me.

Josh Lavin

@jdigory
IRC: digory
Been working with Perl for 13 years.
New to Dancer -- have a lot to learn.

End Point Corporation - web consultancy
Web Application Developer
both front-end and back-end

Lots of older platforms out there. Unavoidable.
Eventually, old apps are migrated.
Or they die (one day; or the last developer maintaining it dies).
Often they need complete rewrites.

Migrate

I wanted to get my legacy app ready to migrate to Dancer.
If I'm going to use modern Perl, I use should tests, right?
Let's write some tests.

Wait, what?
Testing requires methods?
So, _that's_ what Object Oriented means.
Realization my life as a Perl programmer up till now was just doing scripting.
My code was just a big pile; no methods; not testable (except for manual testing of the website).
Business logic everywhere.

Started writing OO Perl, using Moo.
since Dancer2 uses Moo.
Started writing unit tests.
Breaking problems down into smaller problems.
Writing modules was fun!

Hooked up modules to my legacy app.
Oh wait, if my modules depend on legacy app code, I can't run tests from the shell.
(Legacy app can only be called from shell via weird hacks.)
Now I have to abstract away all *Legacy App-specific code* from my modules, use Perl + CPAN modules only (just a couple modules rely on Legacy App).
Now I can run tests.

Move already

Should have moved earlier.
Weird hacks to get Legacy App doing things like Dancer would.
But it was a start.

Now for the front-end

```
@_TOP_@
<h1>[scratch page_title]</h1>
[perl]
   my $has_course;
   for (grep {$_->{mv_ib} eq 'course'} @$Items) {
      $has_course++;
   }
   return $has_course ? '<p>You have a course!</p>' : '';
[/perl]
<button>Buy [if cgi items]more[else]now[/else][/if]</button>
@_BOTTOM_@
```

Legacy app allowed embedding all sorts of stuff into HTML page.
Legacy App tags
+ "Embedded Perl"
= Tag soup.
Won't look nice when viewed on your own machine in a web browser, absent the Legacy App.

Separation of Concerns

HTML + placeholders?
I'm going to use modern Perl, I should separate things, right?
How?
Thought: what if we abstract away logic from the page?
Could we use HTML and placeholders to fill in with our data?
Late to the party, but better late than never.

```
@_TOP_@
[my-tag-attr-list
    page_title="[scratch page_title]"
    has_course="[perl] ... [/perl]"
    buy_phrase="Buy [if cgi items]more[else]now[/else][/if]"
]

    <h1>{PAGE_TITLE}</h1>
    {HAS_COURSE?}<p>You have a course!</p>{/HAS_COURSE?}
    <button>{BUY_PHRASE}</button>

[/my-tag-attr-list]
@_BOTTOM_@
```

First attempt was using Legacy App's built-in placeholder system.
At least the logic was separated from the HTML a bit now.
It worked OK.
But how could we be more ready for Dancer?
(should have just migrated!)
Template::Toolkit?
But hard to use in Legacy App.
Thought: Is it really proper to use conditionals in front-end HTML? (maybe, maybe not)

JavaScript framework
Front-end separated from back-end
Eats JSON
What is this AngularJS everyone keeps talking about?
[next]
Back-end can deliver JSON to front-end.
AngularJS displays data.
As if your front-end is consuming an API!

```
@_TOP_@
<h1 ng-bind="page.title"></h1>
<p ng-if="items.course">You have a course!</p>
<button ng-show="items">Buy more</button>
<button ng-hide="items">Buy now</button>
@_BOTTOM_@
```

After using Angular, now my page looked like this.
(not showing JavaScript)
Now all front-end has from Legacy App is basically "includes" to get header/footer.
The rest is HTML code.
JS handles how the JSON feeds are then displayed in the page.
Cleaner.
Still using Legacy back-end, but all it has to do is "routing" to call the right module and deliver JSON (and do authentication).

Migration to Dancer is now much easier.
I gave it a whirl with a handful of routes and modules.
It went great.

Before:

```
sub _route_feedback {
    my $self = shift;
    my (undef, $sub_action, $code) = split '/', $self->route;
    $code ||= $sub_action;
    $self->_set_status('400 Bad Request');   # start with 400
    my $feedback = MyApp::Feedback->new;
    for ($sub_action) {
        when ("list") {
            my $feedbacks = $feedback->list($code);
            $self->_set_tmp( to_json($feedbacks) );
            $self->_set_path('special/json');
            $self->_set_content_type('application/json; charset=UTF-8');
            $self->_set_status('200 OK') if $feedbacks;
        }
        default {
            for ($self->method) {
                when ('GET') {
                    my $row = $feedback->get($code)
                        or return $self->_route_error;
                    $self->_set_tmp( to_json($row) );
                    $self->_set_path('special/json');
                    $self->_set_content_type('application/json; charset=UTF-8');
                    $self->_set_status('200 OK') if $row;
                }
                when ('POST') {
                    my $params = $self->body_parameters
                        or return $self->_route_error;
                    $params = from_json($params);
                    my $result = $feedback->save($params);
                    $self->_set_status('200 OK') if $result;
                    $self->_set_path('special/json');
                    $self->_set_content_type('application/json; charset=UTF-8');
                }
            }
        }
    }
}
```

This is my Legacy App's route for displaying and accepting feedback entries.
Does not include any authentication checks, nor urldecoding.
This handles feeding back an array of entries for an item ('list'), a single entry (GET), and saving an entry (POST).

After:

```
prefix '/feedback' => sub {
    my $feedback = MyApp::Feedback->new;
    get '/list/:id' => sub {
        return $feedback->list( param 'id' );
    };
    get '/:code' => sub {
        return $feedback->get( param 'code' );
    };
    post '' => sub {
        return $feedback->save( scalar params );
    };
};
```

Dancer gives me a lot for free.
A lot simpler.
Still no auth here, but everything else is done (I can use an auth plugin).

For the front-end, we have options on how to use Dancer:

- Either have Dancer deliver the AngularJS-driven HTML files.
- Or we can have our web server do that, and Dancer deliver the back-end.

Especially if we use Angular as a Single Page App.
We could use JSON Web Tokens (JWT, "jot") for authentication.

Now starring Dancer

In hindsight, should have just moved to Dancer right away.
Legacy App was a pain.
I built my own Routing module in Legacy App...
I built my own Auth checking module, akin to Dancer::PAE...
Dancer makes it simpler.

Dancer is better?

I learned something...

You can do Dancer "wrong".
You can write tag soup in anything, even the best modern tools.
You can do AngularJS "wrong".
I probably do many things the wrong way.
Start somewhere.
Maybe you can't write tests in everything, but you can write smart code.

Dancer is better:

Routes should contain code specific to the Web.
Call non-Dancer modules, where business logic lives.
Return data in the appropriate format.
Dancer is better, when...
(thanks to mst for these)
Makes it easy to test.
Talk to your back-end as if it's an API. Because it is.

Lessons

Separate concerns
Keep it testable
Just start somewhere

The ~~End~~ Beginning

Comments, questions?