HTML::FormFu

Efficient web forms.
So you can leave at 5pm each day.

Dean Hamstead
dean@bytefoundry.com.au

Web frameworks like Catalyst, Mojolicious & Dancer
take care of repetitive things in web app development such as:

- Connecting your code to an environment
  - mod_perl1/2, cgi, fastcgi, nginx_perl etc.
- Mapping URI requests to code
- Authentication, Authorization, Sessions
- Connecting to Databases & ORM's
- Serialization/Deserialization (JSON, XML etc)
- Templating
  - Often built on TT, Alloy etc
- Logging/Debugging levels
  - Often built on Log4perl or Log::Dispatch
- SiteMaps, Caching, Browser detection
- Code Bootstrappers

Whats missing?

Form generation and validation.

- Very rare not to have form’s in software - business applications have lots of them.
- Forms are very boring and repetitive, prone to mistakes and bugs as a result.
- Field data types are frequenly the same things.
  - names, addresses, emails, phone numbers, etc.
- State often isn’t tracked properly, making forms vulernable to attacks / mistakes.
- Reliance on client side javascript only to enforce form data integrity is insufficient
  - Inconsistent browser handling of js makes this more fun

HTML::FormFu

A solution that implements a framework and lots of helpers to handle forms.

Simple define your form using a ’DSL’ of sorts.

(It uses Config::Any so maybe its not a DSL?)

Plug code in to the various events where things need to happen.

Alternatives deserve a mention:

- CGI::FormBuilder
  - even has a Padre plugin!
- HTML::FormHandler
- WWW::Form?
- Others?

Try them and see what suits you!

HTML::FormFu is a stand alone module...

in this presentation I will present it via the Catalyst::Controller::HTML::FormFu module.

Find it all on the CPAN.

- cpanm HTML::FormFu
- cpanm Catalyst::Controller::HTML::FormFu
- apt-get install libhtml-formfu-perl libcatalyst-controller-html-formfu-perl

Examples taken from ’PhotoGame’

- A more than trivial but still small Catalyst app
  - (which I authored)
- Catalyst + MySQL using HTML::FormFu and Imager
  - https://github.com/djzort/PhotoGame
- Basically a "hot or not" voting game clone
- Used to encourage people to take and upload photos at our LANparty events (so we can use them later!)
- Provides a ’tournament’ for photography enthusiasts

How PhotoGame works...

- People register and upload photos
- Uploads are saved to disk, then resized in a cron batch
- Photos can be voted on (A>B or B>A)
  - once per A+B combination per user
- Presents a leader board
- Gradually a winner emerges
- Runs just fine on an early Intel Atom

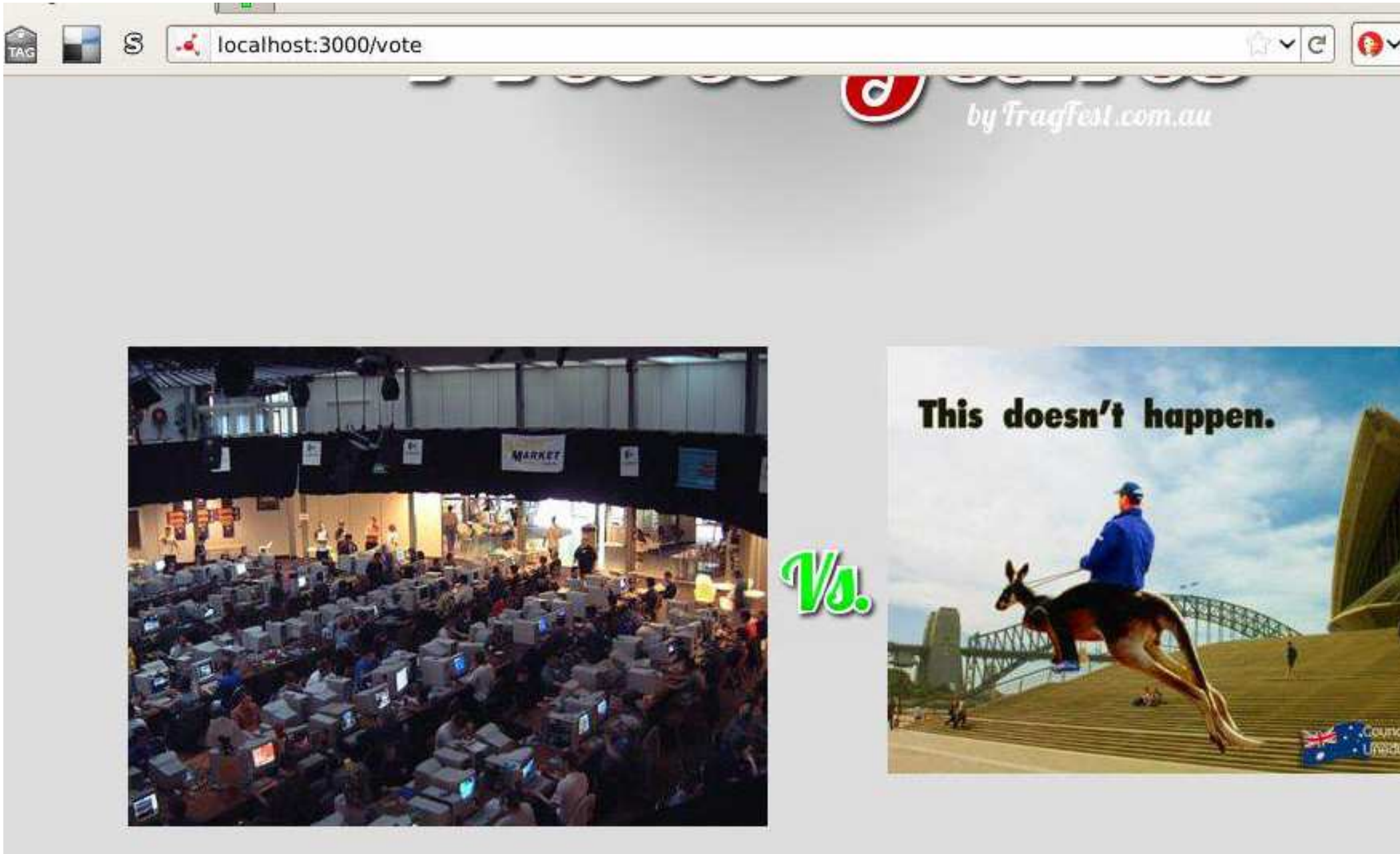
Screenshots...

Screenshots...

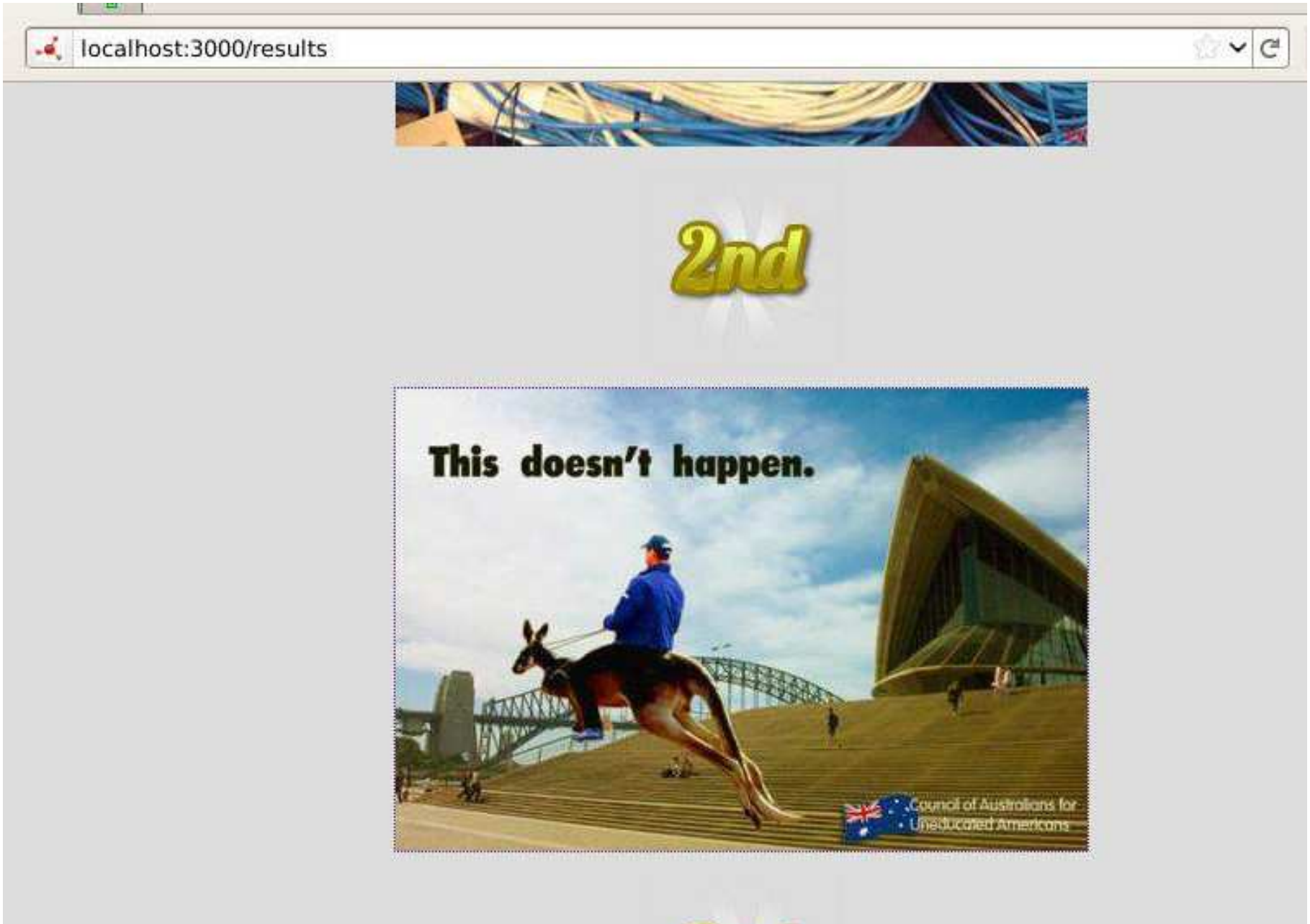
Screenshots...

Screenshots...

Screenshots...



Screenshots...



Getting started

You can use the helper...

- script/myapp_create.pl HTML::FormFu forms

Which will create the ’root/forms’ directory where form configs will go

This default location can be changed via config

(See also perldoc Catalyst::Helper::HTML::FormFu)

Getting started

Your controller can be converted to using FormFu by changing the extends line to this...

- BEGIN { extends ’Catalyst::Controller::HTML::FormFu’ }

Everything will work as normal, until you add the ’FormConfig’ attribute to a subroutine

(or the ’Form’ or ’FormMethod’ attributes)

HTML::FormFu can output either HTML or Template Toolkit streams - which are then processed by a TT based view.

Getting started

Configure as you would expect in Catalyst:

```
$c->config( ’Controller::HTML::FormFu’ => \%my_values );
```

or

```
MyApp->config( ’Controller::HTML::FormFu’ => \%my_values );
```

or, in myapp.conf

```
<Controller::HTML::FormFu>
  default_action_use_path 1
</Controller::HTML::FormFu>
```

Form Validation

Form field input goes through 4 stages, in order, and as long as the result of each stage is successful:

- filters - e.g. remove leading/trailing spaces
- constraints - low-level checks
  - "is this a number?", "is this a valid e-mail address?"
- inflators - turn a value into an object, ready to be passed to validators
- validators - application-level checks
  - "is this username unique?"

See also http://wiki.catalystframework.org/wiki/howtos/forms/formfu.view

Using FormFu in Catalyst

You have two options:

- use the $c->stash->{form} object methods, or
- 'Special Action Name' subs

In either case you add the 'FormConfig' sub attribute.

Object Methods

Unload the HTML::FormFu object with $form = $c->stash->{form}

- $form->submitted
  - true if form submitted
- $form->has_errors
  - true if form has validation errors
- $form->submitted_and_valid
  - shorthand for $form->submitted && !$form->has_errors
- $form->param/s, $form->get_error/s
  - as per perldoc HTMl::FormFu

```
if ($form->submitted_and_valid) {
  #do something
  return 1
}
elsif ( ! $form->submitted ) {
  # defaults
  return 1
}
else {
  # error handling
}
```

Special Action Names

- _FORM_VALID
  - Run when the form has been submitted and has no errors.
- _FORM_SUBMITTED
  - Run when the form has been submitted, regardless of whether or not there was errors.
- _FORM_COMPLETE
  - For MultiForms, is run if the MultiForm is completed.
- _FORM_NOT_VALID
  - Run when the form has been submitted and there were errors.
- _FORM_NOT_SUBMITTED
  - Run when the form has not been submitted.
- _FORM_NOT_COMPLETE
  - For MultiForms, is run if the MultiForm is not completed.
- _FORM_RENDER
  - For normal Form base classes, this subroutine is run after any of the other special methods, unless $form->submitted_and_valid is true.
  - For MultiForm base classes, this subroutine is run after any of the other special methods, unless $multi->complete is true.

Lets use ’Special Action Name’ subs to make a login form.

Here is our perl code...

```
sub login : Path('login') : Args(0) : FormConfig {
  my ( $self, $c ) = @_;
  if ( my $me = $c->session->{iam} ) {
    $c->stash->{message} =
      sprintf( 'You\'re already logged in as %s, go play!', $me->{full_name} );
    $c->detach('message');
  }
}

sub login_FORM_RENDER { }

sub login_FORM_NOT_SUBMITTED { }

sub login_FORM_VALID {
  my ( $self, $c ) = @_;
  if ( my $me =
          $c->model('DB')->check_user(
              $c->req->param('username'),
              $c->req->param('password')
    ) )
  {
    $c->session->{iam} = $me;
    $c->stash->{message} =
        sprintf( 'Welcome %s, lets play!', $me->{full_name} );
        $c->detach('message');
  }
  else {
    $c->stash->{error} = 'Failed to log in';
  }
}
```

- Now lets create the form itself.
- Defined in a config file, named based upon the sub name
- FormFu uses Config::Any - So use a format that suits you
  - YAML, JSON, Perl, etc.
- (I have no idea why I used YAML)
- Example follows

root/forms/login.yml

```
---
action: /login
indicator: submit

elements:
  - type: Fieldset
    name: login
    elements:
      - type: Src
        content_xml: "<legend>Login</legend>"
      - type: Text
        name: username
        label: "Username:"
        add_label_attributes:
            class: auto
        attributes:
          id: username
          title: Enter your username
        constraints:
          - Required
      - type: Password
        name: password
        label: "Password:"
        add_label_attributes:
            class: auto
        attributes:
          id: password
          title: Enter your password
        constraints:
          - Required
      - type: Submit
```

The format/syntax more or less follows what the HTML will look like.

- <form> encompasses a <fieldset> (if you like fieldsets)
- I had to use the 'Src' tag to manually add xml (html) to add a <legend> tag
- <text> for username and password then a submit button
  - These are all implemented in HTML::FormFu::Element::* modules
- Various attributes are defined for each. id, title, class etc

- Constraints are how we tell FormFu how data should be validated
- These are all implemented in HTML::FormFu::Constraint::* modules

- Required means a field must have a value.
  - SingleValue ensures multiple values are not submitted.

Out of the box, you get just about everything you can imagine.
You can easily add more.

- text, textarea, hidden, password
- radio, select, checkbox,
- image, file
- submit/reset buttons.
- derivatives such as email, date, datetime, url

- html elements like hr, fieldset, Src

- constraints for min, max, length, min & max length, number, range
- regex, ascii, email, datetime, word, printable characters,
- set, singlevalue, bool, equal

Plus many many others (homework for you).

Observation...

- Elements and Constraints are role based
- This makes extending things easy
- But you often have to chase through 2-3 levels of modules’s perldoc to work out all available options

Here is a more complex example in the registration form...

```
sub register : Path('register') : Args(0) : FormConfig {
  my ( $self, $c ) = @_;
  unless ( $c->model('DB')->get_setting('registration_open') ) {
    $c->stash->{message} = 'Registrations are closed';
    $c->detach('message');
  }
}

sub register_FORM_RENDER {
}
sub register_FORM_NOT_SUBMITTED {
}

sub register_FORM_VALID {
  my ( $self, $c ) = @_;
  $c->model('DB')->create_photographer(
    username    => $c->req->param('username'),
    password    => $c->req->param('password'),
    full_name   => $c->req->param('full_name'),
    email_addr  => $c->req->param('email_addr'),
    creation_ip => $c->req->address,
  );
  $c->stash->{message} = 'user created';
  $c->detach('message');
}

sub register_FORM_NOT_VALID {
  my ( $self, $c ) = @_;
  $c->stash->{error} = 'Submission failed, see comments above';
}
```

config is root/forms/register.yml

```
---
action: /register
indicator: submit
output_processors:
  - Indent

elements:
  - type: Fieldset
    name: register
    elements:
      - type: Src
        content_xml: "<legend>Register to Compete!</legend>"

      - type: Text
        name: full_name
        label: "Full Name:"
        attributes:
          id: fullname
          title: Full Name
        constraints:
          - type: Required

      - type: Text
        name: email_addr
        label: "Email:"
        attributes:
          id: emailaddr
          title: Email Address
```

```
- type: Text
  name: username
  label: "Username:"
  attributes:
    id: username
    title: Username
  constraints:
    - type: Required
    - type: Length
      min: 2
      max: 255
  validators:
    - '+PhotoGame::Validator::UniqueUsername'

- type: Password
  name: password
  label: "Password:"
  attributes:
    id: password
    title: Password
  constraints:
    - type: Required
    - type: Length
      min: 6
      max: 255
    - type: Equal
      others: repeat-password

- type: Password
  name: repeat-password
```

```
      - type: Submit
        name: submit
        value: Go

constraints:
  - SingleValue
```

The registration form has lots of examples of what can be used.

- ’Required’ constraing in all cases
- ’Email’ constraint used on email field
- ’Min’ and ’Max’ lenfth used on username and password field
- ’Equal’ to ensure sameness of dual password fields

- ’Src’ and ’Hr’ used for a bit of formatting

- Output processor of ’Indent’ makes the HTML look pretty
  - adds cpu, but soothes perfectionist urges :)

- Custom constraint ’+PhotoGame::Validator::UniqueUsername’
  - which checks the DB for username uniqueness

Calls in to the model to check if the username is already used.

Dies with a HTML::FormFu::Excepton object

PhotoGame/Validator/UniqueUsername.pm

```
package PhotoGame::Validator::UniqueUsername;
use strict;
use warnings;
use base 'HTML::FormFu::Validator'; # use parent might be better?

sub validate_value {
  my ( $self, $value, $params ) = @_;
  my $c = $self->form->stash->{context};
  if ($c->model('DB')->username_taken($value)) {
    die HTML::FormFu::Exception::Validator->new({
      message => 'username taken',
    });
  }
  return 1
}
1
```

As you can see its very easy to add/edit/remove form elements.
It’s easy enough that junior or non-programmers can get it right.

Lots and lots of boring code is gone,
which is still good even if you use YAML.

Web Designers can whip up form modifications for clients and finalize them, before handing them to developers to tie in to the database (an ORM may avoid that step entirely)

Using common type constraints will give you consistency through out your software (instead of email being a different regex in different locations, sometimes with MX look ups)

HTML::FormFu may add more CPU usage,
depending on how good/bad your coding is.

You can do cool AutoCRUD stuff with FormFu and DBIC.
A talk for another day.

HTML::FormFu::ExtJS exists.
Promising to generate Javascript forms for you based on ExtJS.

Questions?

ENDE