Server Side Input Validation with Data::FormValidator

Jason Purdy

jason@journalistic.com

Introduction

- Updated Slides will be made available
- Ask Questions Throughout

Trust the User...

WHEEE!!!!

Don't Trust Their Data!

I'm jumping 50!

or...

I can jump
10 more!

The Problem

- Solid User Input
  - Reusable
  - Not Dependent on JavaScript

The (Slightly-Biased) Solution

- Data::FormValidator (aka dfv)

Framework

```
use Data::FormValidator;
use CGI;
# ... some other code here ...
# This method is called when the user hits the
# submit button from an HTML form.
sub process {
     my ( $cgi, $dfv_profile, $results );

     $cgi = new CGI;
     $dfv_profile =
          {
          };
     $results = Data::FormValidator->check(
               $cgi, $dfv_profile
          );
     if (
          $results->has_invalid or
          $results->has_missing
        ) {
          # something's wrong, which you can
          # access what exactly from the
          # $results object
     } else {
          # the user provided complete and valid
          # data ... it's cool to proceed
     }
}
```

Example

- Fields are prefilled with fieldnames
- Checkbox name is 'over18'
- 'Subscribe' button will call `process()`

Required Fields

- `fname`
- `lname`
- `email`
- `password1`
- `password2`

$dfv_profile Update

```
$dfv_profile =
   {
      'required' => [ qw(
                fname lname email
                password1 password2
                ) ],
   };
```

Framework Update

```
# ... [snip] ...
sub process {
    $dfv_profile =
        {
        'required' => [ qw(
                    fname lname email
                    password1 password2
                   ) ],
        };
    $results = Data::FormValidator->check(
            $cgi, $dfv_profile
        );
    if (
        $results->has_invalid or
        $results->has_missing
       ) {
## HERE WE ARE!! ##
    } else {
    }
}
```

What to do w/ Errors?

- Depends (of course)

HTML::Template Approach

```
<html>
<head>
     <title>Subscription Results</title>
</head>
<body>
<h2>Subscription Results</h2>
<!-- TMPL_IF NAME="some_errors" -->
<p>I'm sorry, but there were some problems with your subscription:</p>
<ul>
     <!-- TMPL_IF NAME="err_fname" -->
     <li>You missed your first name.</li>
     <!-- /TMPL_IF -->
     <!-- TMPL_IF NAME="err_lname" -->
     <li>You missed your last name.</li>
     <!-- /TMPL_IF -->
 ... AND SO ON FOR EACH FIELD NAME ...
</ul>
<p>RESULTS Data Structure:</p>
<pre>
     <!-- TMPL_VAR NAME="results" -->
</pre>
<!-- TMPL_ELSE -->
<p>Your subscription has been processed.</p>
<p><a href="/">Back to the Home Page?</a></p>
<!-- /TMPL_IF -->
</body>
</html>
```

Framework Update

```
sub process {
    $template = HTML::Template->new(
            'filename'           => 'results.TMPL',
            'die_on_bad_params' => 0,
          );
    if (
     $results->has_invalid or
     $results->has_missing
    ) {
     my $res_dump = Dumper( $results );
     $template->param(
             'some_errors'   => 1,
             'results'       => $res_dump,
          );
    } else { # valid output }
    print $template->output;
}
```

$results->msgs

- Streamlines error handling

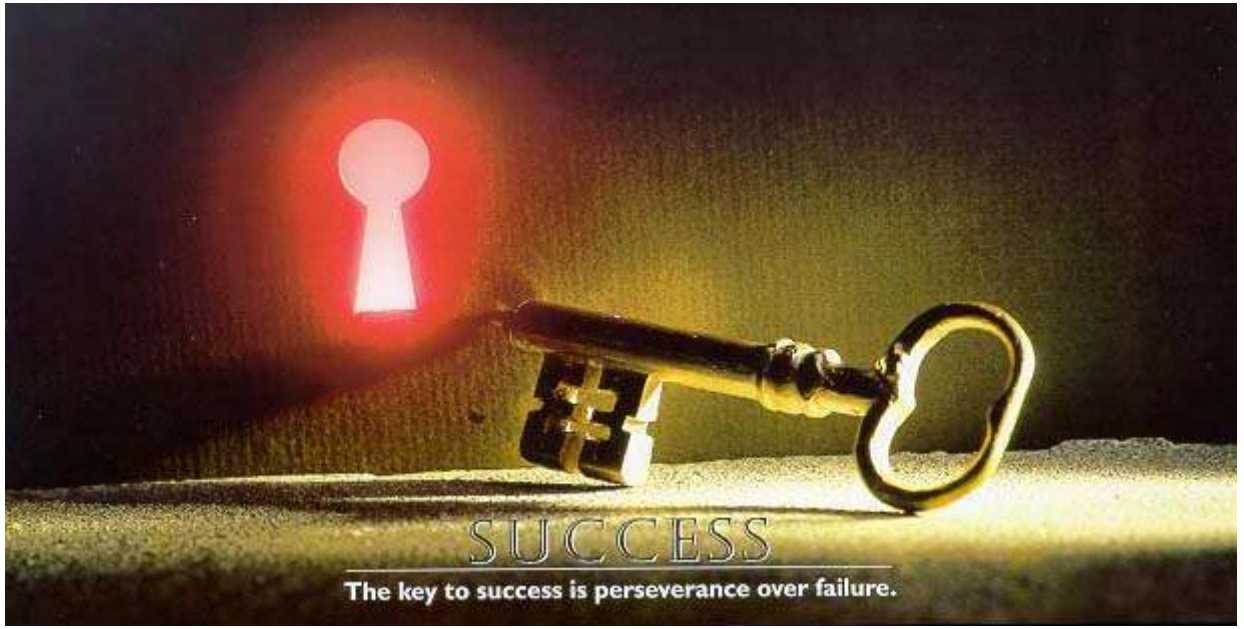
Update $dfv_profile

```
$dfv_profile =
    {
        'required' # ... [snip] ...
        'msgs'     => {
                          'prefix'          => 'err_',
                          'any_errors'      => 'some_errors',
                      },
    };
```

Update Framework

```
sub process {
   # ... [snip] ...
   if (
     $results->has_invalid or
     $results->has_missing
    ) {
     $template->param( $results->msgs );
     my $res_dump = Dumper( $results );
     $template->param(
              'results'        => $res_dump,
         );
   } else { # valid output }
   print $template->output;
}
```

!! Point of Success !!



Email Validation, You Say?

- Constraints are your game

Constraints

- Five Flavor Choices:
  - Built-in
  - Regular Expression
  - Subroutine Reference
  - Hash Reference
  - Array Reference

Email is Built-in

```
$dfv_profile =
    {
     'required'    # ... [snip] ...
     'msgs'        # ... [snip] ...
     'constraints' => {
                       'email' => 'email',
                      },
    };
```

- Loose RFC Check
- Invalid will trigger same framework piece w/ `some_errors` and `err_email`
- Check `Data::FormValidator::Constraints` for more

Could also use the RegEx

```
$dfv_profile =
    {
      'required'    # ... [snip] ...
      'msgs'        # ... [snip] ...
      'constraints' => {
           'email' => qr/\@example\.com$/,
        },
    };
```

- Makes sure that `email` ends in `example.com`

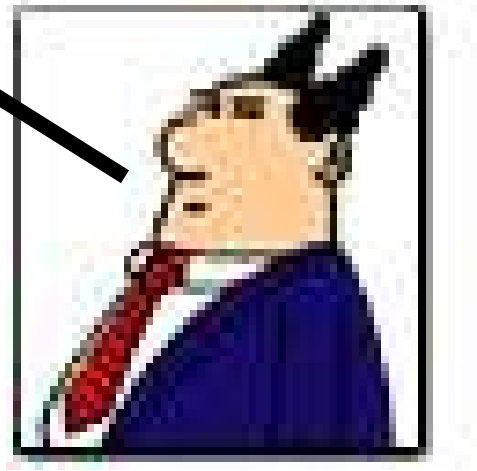
... Or a Subroutine Reference

```
$dfv_profile =
    {
      'required'    # ... [snip] ...
      'msgs'        # ... [snip] ...
      'constraints' => {
           'email' =>
      sub { my $e = shift; return Email::Valid->address($e); },
         },
    };
```

- Verifies address with CPAN Module `Email::Valid`

Uh Oh...

How about making sure the passwords are equal?



Constraints to the Rescue (Again)

```
$dfv_profile =
    {
      'required'    # ... [snip] ...
      'msgs'        # ... [snip] ...
      'constraints' => {
           'password1' => {
                'constraint' => 'check_passwords',
                'params'     => [ qw( password1 password2 ) ],
                     },
                },
    };
```

- **Based on multiple parameters (`password1` and `password2`)**

Just need a `check_passwords()`

```
sub check_passwords
{
    my ( $pw1, $pw2 ) = @_;
    if ( $pw1 eq $pw2 ) {
        return 1;
    } else {
        return 0;
    }
}
```

Constraints Constraint

```
$dfv_profile =
    {
      'required'    => [ qw( fname lname password1 ... ) ],
      'optional'    => [ qw( email ) ],
      'msgs'          # ... [snip] ...
      'constraints' # ... [snip] ...
    };
```

- **Only checked on required or optional fields.**

Lazy Required/Optionals

```
$dfv_profile =
    {
      'required'          => [ qw( fname lname ... ) ],
      'optional'          => [ qw( email ) ],
      'optional_regexp' => qr/password\d/,
    };
```

- A shortcut, in the case of optional passwords
- Also, `required_regexp`

```
$dfv_profile =
    {
          'required'          => [ qw( ... ) ],
          'required_regexp' => qr/\wname/,
    };
```

Dependencies

- If `over18` is checked, then require `dob`

Easy Enough...

```
$dfv_profile =
    {
      'required'     # ... [snip] ...
      'optional'     => [ qw( over18 ) ],
      'dependencies' => {
                           'over18' => [ qw( dob ) ],
                        },
      'msgs'         # ... [snip] ...
      'constraints'  # ... [snip] ...
    };
```

Dependency Groups

```
$dfv_profile =
    {
      'required'           # ... [snip] ...
      'optional'           # ... [snip] ...
      'dependency_groups' => {
                               'passwds' =>
                               [ qw( password1 password2 ) ],
                             },
      'msgs'               # ... [snip] ...
      'constraints'        # ... [snip] ...
    };
```

- **Editing existing profile and passwords aren't required in this case, unless they want to reset the password. In that case, both** `password1` **and** `password2` **are required.**

Other Neat Features...

- Filters
- ConstraintFactory

Future of dfv

- Class::DBI::FormBuilder and CGI::FormBuilder looking to use dfv
  - DFV is looking to build automatic profiles dynamically based on table fields
    - Required and Optional
    - Data types

Parting Nuggets

- Easier to incorporate from the get-go
- Start simple, then evolve
- Useful for other purposes
  - API's
  - Command-Line Scripts
- Subscribe to Mailing List:

  http://lists.sourceforge.net/lists/listinfo/cascade-dataform