

## Example Zend Framework Blog Application Tutorial - Part 6: Introduction to Zend\_Form and Authentication with Zend\_Auth

In the previous entry, we created a new Administration Module to hold blog management functionality, added a Module specific layout for it, and discussed the upcoming need to ensure this is only accessible by authorised Authors. In this entry I'll unravel some of Zend\_Form's mysteries in adding a login form, before using Zend\_Auth to implement authentication for authors.

Previously: [Part 5: Creating Models with Zend\\_Db and adding an Administration Module](#)

Authentication in the Zend Framework is the domain of the Zend\_Auth component, and it is really easy to use. Zend\_Auth is really an abstract API to a number of components working in concert, and without the usual micromanagement of database interaction, sessions, cookies and user data persistence, it makes my life a lot simpler. Of course authentication demands a login form, and so I'll first visit using Zend\_Form. Zend\_Form is an interesting component because it's one of the worst to get started with. The manual, as it does for all components, does not impose a best practice to setting up forms. Mix that with the number of form organisations possible (class based, config based, view template based) and it can be very confusing.

### Step 1: Adding a Login Action and View

Before we actually perform authentication, we need a login form. I've decided to attach all Author account actions to an Author Controller. Add a new file called `AuthorController.php` in `/application/controllers/` containing the following:

```
<?php
class AuthorController extends Zend_Controller_Action
{
    public function loginAction()
    {
    }
    public function logoutAction()
    {
        $this->_forward('index', 'index');
    }
}
```

The logout action for the moment does nothing, but forwards `/author/logout` requests to the main index, just as I would intend to occur after a real logout.


We'll also add a matching template at `/application/views/scripts/author/login.phtml`:

```
<h2>Authentication</h2>
<p>Enter your author name and password below.</p>
<?php echo $this->loginForm ?>
```

Nothing major here, except for a mysterious reference to a view variable, \$loginForm!


## Step 2: Creating a Login form with Zend\_Form

Zend\_Form is one of the most recent additions to the Zend Framework with the release of 1.5. It's not surprising it took so long since a decent Form library is not a trivial component to get through development.

The object oriented approach to developing forms takes a bit of getting used to but it works wonders for simple forms that don't need a heavy design hand. I suppose from my own perspective it was design over functionality that first struck me as problematic when I started using Zend\_Form but I think I'm over that learning curve, so let's see how this look at a simple two field login form goes 

I've deliberately selected a preferred form style to adhere to so this will necessitate customising Zend\_Form options and decorators. It's standard based, tableless, composed of semantic markup, and still looks okay without CSS styling or when using a screenreader (which is one of the more important facets for a form in my opinion).

Like a lot of areas in the Zend Framework, actually organising Form objects is left to your imagination. My first question when approaching any potential object nearly always concerns how reusable I can make it. A reusable form object assumes I'll end up implementing a standard subclass of Zend\_Form so I don't have to repeat myself a dozen times in concrete classes. Hopefully this section provides you with a few good ideas - I have seen Zend\_Form examples in the wild that are horrific so I will spend a chunk of time on Zend\_Form on this outing.

Since I don't intend on mucking about with forms using the traditional design form, apply filtering, extract clean data, process data, re-add data and errors to form template, blah, blah, blah if I can avoid it - I'll use Zend\_Form for almost every form in Maugrim's Marvelous Blog application. Besides, the only public facing form for now would be for comments 

Here's the proposed output I'd be seeking with all this:

```
<form action="/author/login">
<fieldset>
  <legend>Author Authentication</legend>
  <ol>
    <li>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name" />
    </li>
    <li>
      <label for="password">Password:</label>
      <input type="password" name="password" id="password" />
    </li>
  </ol>
</fieldset>
<fieldset class="form-button">
  <input type="submit" value="Submit" />
</fieldset>
</form>
```

Let's see if I can kick `Zend_Form` into cooperating with me on generating that! Or something similar at least...

If `Zend_Form` has a flaw, the biggest one is its documentation because like most of the Zend components it doesn't dictate a best practice for organising the final classes. It also appears a little vague at times, but still it does answer a lot of questions if you read it attentive to detail. If something here is just not making sense do ask in the comments or on the Zend Framework mailing lists.

My own take is to take the term "Divide and Conquer" as my motto for dealing with `Zend_Form`. Break down each specific stage, and from there dump each stage into its own class family. Tackling the whole thing up front without some groundwork is a recipe for the most unmaintainable ugly looking `Zend_Form` implementation imaginable.

With forms our divisions to conquer are quite simple to visualise. We have the form elements which are segregated from any hint of presentation and which carry elements of business logic by virtue of them containing validation/filtering logic. We have the form element decoration which surrounds form elements with semantic markup styled by CSS we can write independently. Finally we have overall layout which groups all elements logically.

At it's simplest, this suggests we'll have two sets of classes. One group for form elements, and another for decorating those form elements. Take the suggested markup from earlier. The `li` tags are obviously decorators of the form elements they wrap. The `fieldset` tags pose another difficulty in that they are not specific to a form element group (there are 2 of them, and the placement is purely for presentation) and so we need some sort of form element grouping mechanism to apply decorators to.

The `Zend_Form` component does precisely that. Despite any confusion it may cause you, it does have a very logical setup for grouping and decorating elements.

Decorators will cause you a few headaches but they aren't completely nuts



. The four main standard types are `ViewHelper`, `Label`, `HtmlTag` and `Error`. There are fourteen in total for even more decorating madness. By default, each of these effects the form element by adding to it's markup in a predefined (but usually configurable) way.

For example, you can use `Label` to prepend a form element with some markup. Obviously the name suggests a `label` tag for the form element but any tag is allowable. Or you can use `HtmlTag` to wrap tags around a form element like a set of `li` tags. Hit the manual for a full description of all 14 decorators.

Revisiting our markup, we can suggest a few decorator steps, by working our way out from the form element. The more specific we get here the better.

1. Prepend form element with `label` tags
2. Wrap form element and `label` with `li` tags
3. Prevent application of 1 and 2 to `submit` element
4. Eliminate default decoration of `submit` element

Once we hit this point in the decorator flow, we leave the domain of the individual form element. `ol` and `fieldset` decorate groups of form elements. `Zend_Form` wins again by letting us create Display Groups for decorating groups of form elements.

5. Create display group for each `fieldset`
6. Wrap display group (name, password) with `ol` tags
7. Prepend display group (name, password) with `legend` tags

8. Wrap display group (name, password) with `fieldset` tags
9. Wrap display group (submit) with `fieldset` tags

Hopefully you're getting the picture. Pick apart your form, and take it step by step. There is a very logical flow with `Zend_Form` that makes implementing each step simple once you identify those steps! If you just jump in you will get lost unless you're already very comfortable with `Zend_Form`.

It's again really important that you watch the ordering of decorators - the first decorator added to the stack will be the first applied to the bare bones form element. So each step should be implemented from the inside out in order.

Delving back into PHP, we can now encode a few standard decorator arrays in a `Zend_Form` subclass. We do need to remember we have one exception above - the `submit` element requires special treatment since it's not decorated as an unordered list. In this subclass we'll also set a few defaults for all forms such as setting a new "accept-charset" attribute for the form element, and ensuring the rendered form object is free of any default decoration apart from the wrapping `form` element itself obviously.

Create the file `/library/ZFBlog/Form.php` as part of our slowly growing application specific library.

```
<?php
class ZFBlog_Form extends Zend_Form
{
    protected $_standardElementDecorator = array(
        'ViewHelper',
        array("Label"),
        array('HtmlTag', array('tag'=>'li'))
    );
    protected $_buttonElementDecorator = array(
        'ViewHelper'
    );
    protected $_standardGroupDecorator = array(
        'FormElements',
        array('HtmlTag', array('tag'=>'ol')),
        'Fieldset'
    );
    protected $_buttonGroupDecorator = array(
        'FormElements',
        'Fieldset'
    );
    public function __construct($options = null)
    {
        parent::__construct($options);
        $this->setAttrib('accept-charset', 'UTF-8');
        $this->setDecorators(array(
            'FormElements',
            'Form'
        ));
    }
}
```

The decorator arrays defined as protected properties are quite simple. For example the `standardGroupDecorator` wraps a display group with the HTML `ol` tag, and then wraps this again with a

`fieldset` tag. The "FormElements" decorator is sort of a group dynamic - we want to render these group HTML tags around all the form elements included in this display group. You'll see this decorator used for any decoration of a display group of form elements, including the parent form as a single display block (helps to think of a form as a single parent display group with children).

As another example, the `standardElementDecorator` prepends a form element with a `label` (notably the label content is not defined yet), and wraps both the form element and label in `li` tags. You'll notice the standard group decorator wraps the parent `ol` tags around all of this.

See how easy that was? Once you break down your form markup into a series of specific decoration steps, writing Zend\_Form decorator arrays tailored to those steps is pretty simple.

So we have a platform for making forms. How about something specific - like a Login form?

Create a new file `/library/ZFBlog/Form/AuthorLogin.php`:

```
<?php
class ZFBlog_Form_AuthorLogin extends ZFBlog_Form
{
    public function init()
    {
        $this->setAction('/author/login');
        // Display Group #1 : Credentials
        $this->addElement('text', 'name', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Name:'
        ));
        $this->addElement('password', 'password', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Password:'
        ));
        $this->addDisplayGroup(
            array('name', 'password'), 'authorlogin',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_standardGroupDecorator,
                'legend' => 'Credentials'
            )
        );
        // Display Group #2 : Submit
        $this->addElement('submit', 'submit', array(
            'decorators' => $this->_buttonElementDecorator,
            'label' => 'Submit'
        ));
        $this->addDisplayGroup(
            array('submit'), 'authorloginsubmit',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_buttonGroupDecorator
            )
        );
    }
}
```

How's that for something interesting? Now all we do for an author login form is define the elements, assign a standard or specific decorator array from the parent class, and assign to a display group which also gets a decorator array from the parent class assigned.

The only real sore point here perhaps, is that I'm declaring the text content of labels and legends in the source code. Some bright mind might find it better to stuff these into a configuration file, and maybe link up a Translator for good measure (not covered here but is a documented possibility in the manual), but for now it's enough to work with.

Let's now revise our AuthorController to create this form and pass it to the View.

```
<?php
class AuthorController extends Zend_Controller_Action
{
    public function loginAction()
    {
        $form = new ZFBlog_Form_AuthorLogin;
        $this->view->loginForm = $form;
    }
    public function logoutAction()
    {
    }
}
}
```

Go ahead and open up your browser to <http://zfblog/author/login>.

Here's the exact output I get from Firefox without tinkering with element separator options:

```
<form enctype="application/x-www-form-urlencoded" action="/author/login" accept-charset="UTF-8"
method="post">
<fieldset id="authorlogin"><legend>Credentials</legend>
<ol>
<li><label for="name" class="optional">Name:</label>
<input type="text" name="name" id="name" value=""></li>
<li><label for="password" class="optional">Password:</label>
<input type="password" name="password" id="password" value=""></li></ol></fieldset>
<fieldset id="authorloginsubmit">
<input type="submit" name="submit" id="submit" value="Submit"></fieldset></form>
```

Apart from needing a few newline separators, it's pretty much what I set out to create



. Zend\_Form just added

a few default classes and attribute values.

### Step 3: Adding Validation to Login Form

Zend\_Form isn't just for presentation since you can also make forms self validating. This takes advantage of all the standard validators you might expect to use manually. Let's revisit the ZFBlog\_Form\_AuthorLogin form

class after including some validation rules.


```
<?php
class ZFBlog_Form_AuthorLogin extends ZFBlog_Form
{
    public function init()
    {
        $this->setAction('/author/login');
        // Display Group #1 : Credentials
        $this->addElement('text', 'name', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Name:',
            'validators' => array(
                array('StringLength', false, array(5,20))
            ),
            'required' => true
        ));
        $this->addElement('password', 'password', array(
            'decorators' => $this->_standardElementDecorator,
            'label' => 'Password:',
            'required' => true
        ));
        $this->addDisplayGroup(
            array('name', 'password'), 'authorlogin',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_standardGroupDecorator,
                'legend' => 'Credentials'
            )
        );
        // Display Group #2 : Submit
        $this->addElement('submit', 'submit', array(
            'decorators' => $this->_buttonElementDecorator,
            'label' => 'Submit'
        ));
        $this->addDisplayGroup(
            array('submit'), 'authorloginsubmit',
            array(
                'disableLoadDefaultDecorators' => true,
                'decorators' => $this->_buttonGroupDecorator,
                'class' => 'submit' // fieldset class attribute for some later styling
            )
        );
    }
}
```

Note the additions including a new "required" flag dictating these values are required (i.e. must not be empty) and also a "validator" array giving a string length minimum and maximum for the "name" form element. I also just threw in a class attribute for our `authorloginsubmit` fieldset display group - it'll make styling that fieldset easier.

One thing blatantly missing is error messages - we haven't added any decorators to our form elements to

include error message text somewhere.

## Step 4: Handling Error Messages with a Custom Decorator

We now have another interesting problem on our hands. Assuming validation fails, where will we display error messages? Usually I prefer to stick them into each form element's `label` tag (I like really short error messages 

) whereas by default, remembering that we've now either disabled or overridden `Zend_Form`'s decoration defaults, it's displayed separately by an appending `Error` decorator.

Since we're departing from the norm, it's time to customise how labels are generated.

What we need to do is intercept a label before it's rendered, and append error messages to it's content. Simplest way of doing that is to subclass the `Label` decorator. There is one additional thing to watch out for which is the fact that the standard `Label` decorator (when used to generate a `label` element) makes use of the `FormLabel` View Helper in `Zend_View`. If we intend appending HTML to the label name (as here) we'll need to disable the `FormLabel` Helper's default treatment of escaping the label name.

This points out another interesting nugget to remember. Many form decorators use `Zend_View`'s View Helpers. And any options passed to a decorator, are also made available to the relevant View Helper. So it does pay to know the View Helpers as well as `Zend_Form` - the helpers have another layer of configurable behaviour you might find handy.

About that custom decorator, it's very simple. It just grabs the error messages for any element it's decorating and appends them to the label name wrapped in `strong` tags for styling access with CSS. Create a new file for the decorator at `/library/ZFBlog/Form/Decorator/LabelError.php`:

```
<?php
class ZFBlog_Form_Decorator_LabelError extends Zend_Form_Decorator_Label
{
    public function getLabel()
    {
        $element = $this->getElement();
        $errors = $element->getMessages();
        if (empty($errors)) {
            return parent::getLabel();
        }
        $label = trim($element->getLabel());
        $label .= '<strong>'
            . implode('</strong><br /><strong>', $errors)
            . '</strong>';
        $element->setLabel($label);
        return parent::getLabel();
    }
}
```


Here we're subclassing the `getLabel()` method - easier this way than retyping the whole original `render()` method! All the method does is append the error messages - we refer back to the parent class version of this method afterwards so we don't duplicate any source code from the standard decorator we're extending.

This is great - so let's wrap up by amending our `ZFBlog_Form` class to tell `Zend_Form` where to find our custom decorator (and indeed any future ones), and also add the new `LabelError` decorator to one of our decorator stacks.

```
<?php
class ZFBlog_Form extends Zend_Form
{
    protected $_standardElementDecorator = array(
        'ViewHelper',
        array('LabelError', array('escape'=>false)),
        array('HtmlTag', array('tag'=>'li'))
    );
    protected $_buttonElementDecorator = array(
        'ViewHelper'
    );
    protected $_standardGroupDecorator = array(
        'FormElements',
        array('HtmlTag', array('tag'=>'ol')),
        'Fieldset'
    );
    protected $_buttonGroupDecorator = array(
        'FormElements',
        'Fieldset'
    );
    public function __construct($options = null)
    {
        // Path setting for custom decorations MUST ALWAYS be first!
        $this->addElementPrefixPath('ZFBlog_Form_Decorator', 'ZFBlog/Form/Decorator/', 'decorator');
        parent::__construct($options);
        $this->setAttrib('accept-charset', 'UTF-8');
        $this->setDecorators(array(
            'FormElements',
            'Form'
        ));
    }
}
```


The new decorator reference receives a new option - we disable escaping of the label's name value so any included HTML doesn't get the `htmlspecialchars` treatment. This option is passed into the `FormLabel` View Helper when called from the decorator class.

In our constructor we have a comment. Never, ever, ever, forget that comment. It is essential that decorator paths for your custom decorators are added as early as possible - in fact they should be the first thing you do in a subclass of `Zend_Form` even before passing options to the parent's constructor.

Feel like taking a peek? Go ahead and browse to <http://zfblog/author/login>. The form is still there, and it looks like it hasn't broken yet 

. Go team!

## Step 5: Replacing Those Cumbbersome Default Errors

One final step I'll make is layering in a Translation object for our forms. This has a few purposes. First it will allow for translating labels, legends, etc., but to be honest I'm really doing because it's the simplest method of getting rid of the long error messages the Validators generate. Life is never easy, eh? 

We'll start simple and only address error messages as a quick example. Add a new file at `/translate/forms.php` containing:

```
<?php
return array(
    Zend_Validate_NotEmpty::IS_EMPTY => 'Required',
    Zend_Validate_StringLength::TOO_SHORT => 'Minimum Length of %min%',
    Zend_Validate_StringLength::TOO_LONG => 'Maximum Length of %max%'
);
```

With the translation file in place (using the Zend\_Translate Array adapter) we just need to tell Zend\_Form where to find it. Zend\_Form uses a static method for this, so I'll create a quick static check and helper function for our ZFBlog\_Form class.

```
<?php
class ZFBlog_Form extends Zend_Form
{
    protected $_standardElementDecorator = array(
        'ViewHelper',
        array('LabelError', array('escape'=>false)),
        array('HtmlTag', array('tag'=>'li'))
    );
    protected $_buttonElementDecorator = array(
        'ViewHelper'
    );
    protected $_standardGroupDecorator = array(
        'FormElements',
        array('HtmlTag', array('tag'=>'ol')),
        'Fieldset'
    );
    protected $_buttonGroupDecorator = array(
        'FormElements',
        'Fieldset'
    );
    public function __construct($options = null)
    {
        // Path setting for custom decorations MUST ALWAYS be first!
        $this->addElementPrefixPath('ZFBlog_Form_Decorator', 'ZFBlog/Form/Decorator/', 'decorator');
        $this->_setUpTranslation();
        parent::__construct($options);
        $this->setAttrib('accept-charset', 'UTF-8');
        $this->setDecorators(array(
            'FormElements',
            'Form'
        ));
    }
}
```


```

protected function _setupTranslation()
{
    if (self::getDefaultTranslator()) {
        return;
    }
    $path = dirname(dirname(dirname(<u>_FILE_</u>)))
        . '/translate/forms.php';
    $translate = new Zend_Translate('array', $path, 'en');
    Zend_Form::setDefaultTranslator($translate);
}
}

```

Easy as pie. Now all error messages will be replaced.

## Step 6: Introducing Joe Bloggs

Before we go further, we're missing one essential aspect of Authentication. We don't have a user! Let's resolve that quickly. I'm not going to pounce on registration since a) this is a one-person blog, and b) I can add it later. So I'll use some filler data for a theoretical user instead. I'm cheap I know. Sorry 

Here's Joe. Rumour has it he's The Common Man (TM). If you prefer, Joe can be a Jane. Nobody intends hunting him/her down to check anyway...

```

INSERT INTO `authors` (`realname`, `username`, `password`, `email`) VALUES ('Joe
Bloggs', 'joebloggs',
'5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8',
'joe.bloggs@example.com');

```

The hashed password is "password" hashed using the SHA256 algorithm.

Run the SQL insert on your users table in the database to manually register Joe (or Jane).

## Step 7: Adding Authentication and Form Processing to AuthorController

Zend\_Auth operates by referencing an adapter for our storage system. Since we're using a database, we'll use the `Zend_Auth_Adapter_DbTable` adapter. The adapter works by allowing us to define specific fields on that table used for the login process, i.e. the username and password fields. It also lets us define an operation needed for these fields which in our case is primarily the need to apply SHA256 to any password before comparing it to the database stored value. Unfortunately since MySQL and SHA256 aren't happy campers I've explicitly applied the hashing here.

Once the Adapter is configured we need only add two steps. Firstly we tell the Adapter to attempt authentication using the values provided from our login form, and secondly we store the authenticated author's data for future reference elsewhere in the application.

Since the only authenticated user is the author, on a successful login we'll redirect them to the Administration Module's index page.

Here's the updated version of our AuthorController class.


```
<?php
class AuthorController extends Zend_Controller_Action
{

    public function loginAction()
    {
        $form = new ZFBlog_Form_AuthorLogin;
        if (!$this->getRequest()->isPost() || !$form->isValid($_POST)) {
            $this->view->loginForm = $form;
            return;
        }
        $values = $form->getValues();
        // Setup DbTable adapter
        $adapter = new Zend_Auth_Adapter_DbTable(
            Zend_Db_Table::getDefaultAdapter() // set earlier in Bootstrap
        );
        $adapter->setTableName('authors');
        $adapter->setIdentityColumn('username');
        $adapter->setCredentialColumn('password');
        $adapter->setIdentity($values['name']);
        $adapter->setCredential(
            hash('SHA256', $values['password'])
        );
        // authentication attempt
        $auth = Zend_Auth::getInstance();
        $result = $auth->authenticate($adapter);
        // authentication succeeded
        if ($result->isValid()) {
            $auth->getStorage()
                ->write($adapter->getResultRowObject(null, 'password'));
            $this->_helper->redirector('index', 'index', 'admin');
        } else { // or not! Back to the login page!
            $this->view->failedAuthentication = true;
            $this->view->loginForm = $form;
        }
    }
    public function logoutAction()
    {
        Zend_Auth::getInstance()->clearIdentity();
        $this->_helper->redirector('index', 'index');
    }
}
```

As you can see, using Zend\_Auth is pretty easy. Authentication is driven by the Adapter in use, and performed by a quick call to the singleton (there can only be one such object globally) Zend\_Auth instance. The rest is storing the credentials and setting View result values for our template to make use of.

Note that the getResultRowObject() method parameters tell the storage driver not to remember the password hash. Personally I don't want the hash leaving the database if I can help it - it serves no purpose

and is the kind of user data you might consider not persisting outside authentication. You could drop that entire line if you didn't care if passwords are stored or not since Zend\_Auth will store the identity by default.

I've also added a logout call to `clearIdentity()` which does exactly what it says on the tin 

. Might be

useful when you're testing at home - I'll add a logout link to the interface another time.

A little feedback would probably be appreciated by the author if their login attempt failed, so let's amend our login View at `/application/views/scripts/author/login.phtml` accordingly.

## <h2>Authentication</h2>

<p>Enter your author name and password below.</p>

<?php if ([isset](#)(\$this->failedAuthentication)): ?>

<p class="error">Sorry, but the credentials supplied could not be authenticated. Please try again.</p>

<?php endif; ?>


<?php [echo](#) \$this->loginForm->render() ?>

Go ahead, open a browser and take the new login form for a test run.

## Conclusion

Since authentication is really simple with the Zend\_Auth component, the bulk of this installment concerned using Zend\_Form. I really do advise taking a form apart before calling on Zend\_Form because you really need to know how a form's structure can be built up in layers using the Zend\_Form decorators.

If anything, the decorators are the main obstacle you'll meet using Zend\_Form since almost everything else is straight forward once you put in place a bit of structure. At all costs avoid the umpteen line approach to Zend\_Form usage where elements, decorators and validators are tied so closely together maintenance is a lost cause. One point I didn't cover was adding custom Element classes - in a form heavy application that could save you even more typing.


Our authentication now in place, the next Part will introduce two unrelated areas. Authorisation with Zend\_Acl, followed by a brief detour into applying some styling to the growing bank of HTML (I'm getting a little tired by the CSSless look 

).

Note: The source code for this entry is available to browse, or checkout with subversion, from <http://svn.astrumfutura.org/zfblog/tags/Part6>. The full source code for the entire application (as it exists thus far) from <http://svn.astrumfutura.org/zfblog>. The source code now contains some additional directories utilised to build distributable versions of the source code using Phing.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 06:55

Finally part 6 is here.

Thanks alot. I really appreciate your work and your effort writing these great articles. This series is coming exactly at the right time (at least for me 

) and to be honest I have never waited that impatiently for a blog entry to come (actually I never waited for any blog).

So, keep going to enlighten me.

Cheers,

Ro Anonymous on May 7 2008, 20:38

For those reading early enough, there was a small formatting error in Part 6 which is now resolved so the full entry is visible.

Anonymous on May 7 2008, 21:36

Why do you place authentication in author controller? It should be separated imho. One nice plugin with acl I guess would be nicer.

I know, that all depends on readers



Anonymous on May 8 2008, 04:02

I just stick stuff where it makes the most immediate sense



. A plugin that's reusable would be nice since one is needed for ACL

anyway. Anonymous on May 8 2008, 04:46

Do you have a Paypal/other donate button somewhere on your blog? I can't seem to find it. Forget about publishing something, this series alone is worth the price of a book.

I know some people don't like doing this but everybody has to eat...

Time is money and you've saved me a boat load!

rojo Anonymous on May 8 2008, 05:07

My copy of the Wannabe Jedi Handbook clearly states commentors are not allowed to reveal The Plan. Since it's secret...



But yes, there will be a donation option in the near future. There will also be an updated version of this series in HTML and PDF formats. And it will be available for free under a Creative Commons License with a donation option only.

I suggest taking a look at what's emerging in "docs" in subversion.

I'm an open book, I am... Anonymous on May 8 2008, 05:53

I have to admit that I agree with some of the "too complicated" comments on the recent Zend\_Form Decorators article. I imagine that with time I'll begin to appreciate the flexibility we're afforded, but the learning curve here is steep.

I'd like to insert specific CSS classes into the text and button inputs. It was simple enough to insert a class into the form element, but I have yet to figure how to do the same for the text inputs.

Aside from this, I felt sorry for the lonely, unused logout method so I added `<a href="/author/logout">aLogout</a>` to the admin layout's sidebar in `application/admin/views/layouts/admin.phtml`




Anonymous on May 8 2008, 07:03

Those links didn't come out as expected




Here's the referenced article: [http://devzone.zend.com/article/3450-Decorators-with-Zend\\_Form](http://devzone.zend.com/article/3450-Decorators-with-Zend_Form) Anonymous on May 8 2008, 07:06

You can add class values to an element using the `setClass()` method, or as with the submit display group, a "class" parameter.

Should work 

I added some styling to the form in the subversion trunk a little while ago, and it worked out quite well without any other additions - the `ol/li` tags are usually enough as CSS selectors. Anonymous on May 8 2008, 07:14

Thanks for these wonderful articles! Keep them coming! I have a question on a ZF issue, which is however not directly related to the above. But perhaps you could help me out on this one anyway 

In my own application at some points I'd like to return json using the json action helper. Often, in my action controller, I'm first obtaining a rowset of my model rows (extensions of `zend_db_table_row`). And in fact I then want to use this rowset to prepare an array of data that needs to be json returned. Now where would I put the piece of code that transforms the rowset into a custom array (so not just `rowset->toArray()`) that gets json returned?

I guess, it's in a sense part of the 'view', so should I wrap it in a view helper? What would you do?

Thanks already and again, I'm eagerly waiting for the next installment of this series! Anonymous on May 8 2008, 14:53

There is a little mistake in this line

We'll also add a matching template at `/application/views/author/login.phtml`:

it must be


We'll also add a matching template at `/application/views/*scripts*/author/login.phtml`: Anonymous on May 8 2008, 15:11

I don't like form creator for me the best way is do it in html. Anonymous on May 8 2008, 15:47

It really depends on what appears to be the simpler solution. Assuming the array building is purely for json output, then it's presentation.

That doesn't necessarily mean putting it in a View Helper. If it's only needed by the Json helper in a controller - then ordinarily I'd suggest adding it to the Model as a helper method. If you're not using a Model (`Zend_Db_Table`), I'd suggest an Action Helper.

Don't forget that it might just be as easy to stick into the controller method directly if it's small enough, and used infrequently. Anonymous on May 8 2008, 16:02

Hmm, is there a way to follow comments on this blog without leaving a comment yourself? 


Anonymous on May 8 2008, 18:44

This is the best description of how to use `Zend_Form` I've ever read. This is a very good series to get started with Zend Framework.

And like Roman wrote: For me this is the only Blog, where I am always waiting for the new Parts.

Thank You for Your efforts!! Anonymous on May 8 2008, 18:52

Why do you often use multiple `dirname()` calls? Wouldn't it be clearer and better for the performance if you defined a constant like `ZFBLOG_ROOT` in your bootstrap? Anonymous on May 8 2008, 19:14

Probably, maybe, perhaps... 

. One of the reasons to be honest is to avoid temptation. It's too easy to get so used to a constant it turns up absolutely everywhere. Give me a few more parts and it'll creep in when it's definitively required only in 2-3 places. Anonymous on May 8 2008, 20:11

Unordered lists are the Swiss Army Knife of markup, but I prefer not to use them for form markup. I instead use this method <http://2tbsp.com/node/50>

Hmmm. I had tried a class parameter but `addElement()` applies the class to both the input and label. I made my CSS a bit more specific to apply to just input's and it looks good now.

I'd still like to figure out how to create a decorator for just inputs that could be applied before the `_standardElementDecorator`. Anonymous on May 8 2008, 23:40

I haven't seen it previously (don't add classes a lot) but you're right - it's applied to both the input and label elements. Hmmm. Seems a little presumptuous to be doing that so perhaps an extra recommendation would be ensure you use very specific CSS selectors? Anonymous on May 9 2008, 00:04

This is a really awesome Tutorial for `Zend_form`.

I did run into a problem halfway through step 6 though. I think I've narrowed the source of the problem down to when `AuthorLogin.php` get this added to it:

```
$this->addElement('text', 'name', array(
'decorators' => $this->_standardElementDecorator,
'label' => 'Name:',
'validators' => array(
array('StringLength', false, array(5,20))
),
'required' => true
));
```

The browser returns this error:

Fatal error: Call to undefined method `ReflectionClass::newInstanceArgs()` in `C:\Program Files\Apache Group\Apache2\htdocs\zfblog\library\Zend\Form\Element.php` on line 996

I tried googling around and the only things i found were people with the same problem, still unresolved.

Do i have the wrong version of something or did i just miss something? Anonymous on May 9 2008, 00:43

Does anyone have any idea why to get the css to display properly on all pages I must put `"/public/"` before the path in my href attribute.

Ex.

Instead of:

Anonymous on May 9 2008, 00:55

Have a look at the PHP documentation for the `Reflection` class.

Quote:

"Note: hasConstant(), hasMethod(), hasProperty(), getStaticPropertyValue() and setStaticPropertyValue() were added in PHP 5.1.0, while newInstanceArgs() was added in PHP 5.1.3." Anonymous on May 9 2008, 01:01

My bad, sorry for double post, forgot the code tags.

Does anyone have any idea why to get the css to display properly on all pages I must put "/public/" before the path in my href attribute.

ex

```
href="/public/css/blueprint/screen.css"
```

instead of

```
href="/css/blueprint/screen.css"
```

Damn geshi tags Anonymous on May 9 2008, 01:15

A lot of the href issues are that the base directory for the server isn't set correctly to /path/to/public.

If you are using a virtual domain as described this won't cause any issues. On the other hand if you're using a relative directory from another path, e.g. localhost/zfblog/public, then you will need to prefix all href values with the base path.

Perhaps you need to set a `<base href="localhost/path/to/public" />` tag inside `<head>` in the layout templates - at least then its isolated in one place... After that DO NOT include the beginning forward slash used elsewhere in templates for href values.

Anonymous on May 9 2008, 02:46

What about storing path information in config.ini. What about storing configuration settings with Zend\_Registry instead of constant?

Anonymous on May 9 2008, 04:21

It is interesting the db based authentication is relatively simple with ZF while the Zend Form stuff is... not very user friendly, to put it mildly.

I just can't get over the fact that Zend\_Form seems to be very complicated and clunky. I'm often slow on the uptake but Zend Form gives me more headaches than just about any other aspect of ZF! Ack. Anonymous on May 9 2008, 05:01

Unfortunately all the extra complexity is to cater to a wide variety of needs. The decorators I would agree could do with a renaming - they are really very simple to use but the names and organisation obscure that.

Otherwise consider what you get in return - self validation, ease of customisation, class/config based definition, etc. It's not a bad balance at all. Anonymous on May 9 2008, 05:16

Thanks man,

It's the first really usefull and well explained tutorial I found since I begin learning ZF a month ago.

If in a near future you can add some sessions stuff, it will be great as for now I don't see how to use it without calling it in every controllers or subclassing the controller parent. But as I read before, it seems to be not the right solution, maybe an action helper ?

Anonymous on May 9 2008, 17:57

If using it from within a controller, I'd venture an action helper would be worthwhile. The other facet is to remember a lot of components integrate Zend\_Session behind the scenes (e.g. Zend\_Auth uses it internally so you don't need an external reference).

I'll see if I need Sessions more directly in the future and work in a good example usage. Anonymous on May 9 2008, 19:06

I don't like having path data in a config file, since we can figure it automatically we should do so.

But we set \$root = ""; in the bootstrap file anyway, so why don't we add it to the registry or something? Anonymous on May 10 2008, 02:02

Guys, so far it's a one or two references. If it becomes an issue it can be dealt with. For the moment though there's little to gain... Anonymous on May 10 2008, 02:23

Ok, this is fairly complicated stuff I've always avoided for now - especially because Zend\_Form documentation is a little vague here and there.

But I think I got most of it from your great article, practice will show how much.



Anyway I've collected a few questions while reading the article:

IIRC the Zend\_Form documentation uses a method like getLoginForm() in a controller while you set up an object. Since the login form is probably used only in the AuthorController anyway I'd have used getLoginForm() too. This is probably not "wrong", but would it be bad practice?

I've noticed that Zend\_Form always uses the elements name as id attribute too. Is there a way to avoid that? I'm thinking of pages with e.g. two forms, both having a "send" button.

Don't you dare using a minimum length of 5! (Hint: my nickname



You wrote "The new decorator reference receives a new option - we disable escaping of the label's name value so any included HTML doesn't get the htmlspecialchars treatment." - where does that happen? I'm missing it somehow. Anonymous on May 10 2008, 04:36

We probably just want to know how it's done.

How everything is done perhaps.

Or as my friend Number 5 kept saying: More input, more, more



Anonymous on May 10 2008, 04:40


As regards the object vs method, it's not bad practice per se. I don't know the example you're referencing, but my own opinion is that if it's reusable it belongs in a separate object, not a controller method.

Element IDs default to the name value because that's the convention used by the underlying Zend\_View\_Helper\_FormElement class. To switch any form attribute (note these are often copied to attached decorators like Label) just add a new "attrs" parameter with an array where keys are the attrib name, and values the attrib value. Taking the example of a login form's element for the username:


```
$this->addElement('text', 'name', array(
'decorators' => $this->_standardElementDecorator,
'label' => 'Name:',
'validators' => array(
array('StringLength', false, array(5,20))
),
'required' => true,
```

```
'attrs' => array('id' => 'stfu')
));
```

The option to prevent escaping is passed into a decorator via its parameter array. Since I stuck decorators into ZFBlog\_Form, the standard element decorator uses the LabelError custom decorator with an array of params include the escape option set to false.

A better minimum is 3 I agree 


. Sorry! Anonymous on May 10 2008, 05:36

Why does a login have to be limited to a minimum count anyway? 

I just stumbled over that a few days ago:


<http://thedailywtf.com/Articles/Waiting-by-the-Mailbox.aspx>

3rd picture...

Thanks for answering that fast and the tutorial in total. I've said it before and will say it again: it rocks. 

Anonymous on May 10 2008,

08:00

I've amended the username field in subversion. Now the minimum is 3 characters, and I've added a maxlength attribute of 20 in keeping with the validation rules 

. Anonymous on May 10 2008, 08:25

There is some tweaking that needs to be done for Zend\_Form to display the form the way you intended to (hint: closing of self contained tags like and , as required for XHTML).

In the Bootstrap, method setupView should set the doctype, like so:

```
public static function setupView()
{
    $view = new Zend_View;
    $view->setEncoding('UTF-8');
    $view->doctype('XHTML1_STRICT');
    $viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
    Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
    Zend_Layout::startMvc(
        array(
            'layoutPath' => self::$root . '/application/views/layouts',
            'layout' => 'common',
            'pluginClass' => 'Webote_Layout_Controller_Plugin_Layout'
        )
    );
}
```

XHTML1\_STRICT could be changed, but since we are building it from scratch I don't see any reason not to go with the strict DTD.

This way, Zend\_Form automatically outputs XHTML markup for the forms. Anonymous on May 12 2008, 03:39

Your quite right, but I deliberately have avoided too much Zend\_View detail and deferred it for a more in-depth look in Part 9.



The

compounding errors and duplications are intentional since a flawed example will underline the usefulness of using View Helpers, Zend\_View, et al. to fix them. Anonymous on May 12 2008, 04:17

Great job, these tuts are great for me.

I can't believe that the official tutorial at zend is not more like this. Anonymous on Dec 3 2008, 11:54

The problem with the Zend tutorial is that it's not designed to foster a deeper understanding. It's a quickstart a new visitor to the site can complete as quickly as possible. Sort of an instant gratification to whet the visitors appetite so they are persuaded to stick around a bit longer. Anonymous on Dec 3 2008, 16:54

Strangely enough, i didn't get the messages displayed when i submit the form ..though there is no problem and i followed the steps from 1 till 5 without getting an error, the form is displayed as it should but no error messages..if i left the values of user or password blanks..any helps!!! Anonymous on Dec 28 2008, 01:36

I'm working on formalising everything



. Take a look at the new book site over on <http://www.survivethedeepend.com> Anonymous on

Jan 7 2009, 08:14

Hi,

I'm really enjoying the series so far but i've hit a bit of a brick wall....

I get the error:

```
Fatal error: Uncaught exception 'Zend_Controller_Dispatcher_Exception' with message 'Invalid controller specified (author)' in
/Applications/MAMP/htdocs/zfblog/zf-install/library/Zend/Controller/Dispatcher/Standard.php:241 Stack trace: #0
/Applications/MAMP/htdocs/zfblog/zf-install/library/Zend/Controller/Front.php(934):
Zend_Controller_Dispatcher_Standard->dispatch(Object(Zend_Controller_Request_Http), Object(Zend_Controller_Response_Http))
#1 /Applications/MAMP/htdocs/zfblog/v1/application/Bootstrap.php(17): Zend_Controller_Front->dispatch() #2
/Applications/MAMP/htdocs/zfblog/v1/public/index.php(15): Bootstrap::run() #3 {main} thrown in
/Applications/MAMP/htdocs/zfblog/zf-install/library/Zend/Controller/Dispatcher/Standard.php on line 241
```

I think it means a controller is incorrectly named but I cant find any problems!

Hope someone can help.... Thanks.

Ben Anonymous on Jan 18 2009, 23:49

It looks to me like you dont have an AuthorController.php In your application/controllers directory.

Or that the class is not called 'AuthorController' Anonymous on Jan 21 2009, 18:24

I'm having the same trouble.

I checked that it was definitely calling the correct Label decorator by appending some text whatever and its fine. It looks like its not doing the validation.

```
$this->addElement('text', 'name', array(
    'decorators' => $this->_standardElementDecorator,
    'label' => 'Name:',
    'validators' => array(
        array('StringLength', false, array(5,20))
    ),
    'required' => true,
));
```

Anonymous on Jan 21 2009, 18:42

Hi.

I've actually completed the next step of this tutorial as well but that's because I forgot I was having the problem I'll explain in a mo.

When I try and log in using 'joebloggs' and 'password' a window pops up with the following info: 'Apache HTTP Server has encountered a problem and needs to close. We are sorry for the inconvenience.'

Basically Apache crashes so it don't know if Zend is doing something wrong or I haven't enabled something in Apache.

Does anyone have any idea what I may have missed?

Thanks Anonymous on Jan 31 2009, 23:59

great tutorial...it has brought alot to light Anonymous on Feb 3 2009, 21:44

Hey, loving the series - following along from home and learning a lot. Really dig your work!

Just a suggestion for this article - it could do with a link to the next/7th chapter, I had to go searching via the main page. Anonymous on Feb 7 2009, 00:01