Zend Framework provides a rich and flexible form component built using the object-oriented features of PHP 5, proven design patterns, and a modular architecture.

Matthew Weier O'Phinney
PHP Developer
Zend Technologies
Topics Overview

• What Problem Zend_Form Addresses
• Benefits of Zend_Form
• Overview of Zend_Form's design
• In-Depth Discussion of Zend_Form components
• Using Zend_Form with MVC Applications
• Q & A
The Problem (and the Solution)
Why do we need a Form component?

• What is a form?
  ▪ HTML Markup
  ▪ Input Filtering (Sanitization and Validation)
  ▪ Error Reporting
What the form looks like to the user:
The HTML markup of the form:

```html
<h2>Please Login</h2>
<form action="/login/process" method="post">
  <dl class="zend_form">
    <dt><label for="username">Username</label></dt>
    <dd><input name="username" type="text" id="username" value="" /></dd>
    <dt><label for="password">Password</label></dt>
    <dd><input name="password" type="password" id="password" value="" /></dd>
    <dt></dt>
    <dd><input name="login" type="submit" id="login" value="Login!" class="submit" /></dd>
  </dl>
</form>
```
The Zend_View view script generating the HTML:

```html
<form action="/login/process" method="post">
  <dl class="zend_form">
    <dt><? $this->formLabel('username', 'Username') ?></dt>
    <dd><? $this->formText('username') ?></dd>
    <dt><? $this->formLabel('password', 'Password') ?></dt>
    <dd><? $this->formPassword('password') ?></dd>
    <dd><? $this->formSubmit('login', 'Login!', array('class' => 'submit')) ?></dd>
  </dl>
</form>
```
Manually, using String functions and PCRE:

```php
if (!preg_match('/^[a-z0-9]{3,20}$/i', strtolower(trim($username)))) {
    // Failed
}
if (6 > strlen(trim($password))) {
    // Failed
}
```

- Short and sweet, but not easily configurable
Zend_Filter and Zend_Validate chains:

```php
$usernameFilter = new Zend_Filter();
$usernameFilter->addFilter(new Zend_Filter_StringTrim())
    ->addFilter(new Zend_Filter_StringToLower());
$passwordFilter = new Zend_Filter();
$passwordFilter->addFilter(new Zend_Filter_StringTrim());

$usernameValidator = new Zend_Validate();
$usernameValidator->addValidator(new Zend_Validate_Alnum())
    ->addValidator(new Zend_Validate_StringLength(3, 20));
$passwordValidator = new Zend_Validate();
$passwordValidator->addValidator(new Zend_Validate_StringLength(6));

if (!$usernameValidator->isValid($usernameFilter->filter($username))) {
    // Failed
}
if (!$passwordValidator->isValid($passwordFilter->filter($password))) {
    // Failed
}
```

- Lots of code, and not terribly configurable
Zend_Filter_Input:

- Configurable, but only operates on sets of data. Best so far.

```php
$zfi = new Zend_Filter_Input(
    array(
        '*' => 'StringTrim',
        'username' => 'StringToLowerCase'
    ),
    array(
        'username' => array(
            'Alnum',
            array('StringLength', 3, 20)
        ),
        'password' => array(
            array('StringLength', 6)
        ),
    ),
    $data
);
if (!$zfi->isValid()) {
    // Failed
}
```
Now we need to get error messages into the form. First, we need to get the messages:

- Manual validation: make them up as we go (not standardized, and thus not good for translation)
- Zend_Validate: getMessages()
- Zend_Filter_Input: getMessages() (each element returns an array of messages exactly as Zend_Validate would)
Error Reporting

- Pass messages and values to the view
- In the view script:
  - For each input, check for errors, and conditionally display
  - For each input, check for passed in value and conditionally display
Error Reporting

Example:

```php
<dt><? $this->formLabel('username', 'Username') ?></dt>
<dd><? $this->formText('username', $this->values['username']) ?></dd>
<? if (isset($this->errors) && array_key_exists('username', $this->errors)): ?>
<ul class="errors">
><? foreach ($this->errors['username'] as $error): ?>
    <li><? $this->escape($error) ?></li>
<? endforeach ?></ul>
<? endif ?></dd>
```
We end up duplicating element information between rendering, input filtering, and error reporting.
The Solution: Zend_Form

Create elements: Username:

```php
$username = new Zend_Form_Element_Text('username');
$username->addFilters(array('StringTrim', 'StringToLowerCase'))
    ->addValidators(array('Alpha',
                            array('StringLength', false, array(3, 20))
                          ))
    ->setRequired(true)
    ->setLabel('Username');
```

- Multiple filters (filter chain!)
- Multiple validators (validator chain!)
- Required
- Don't forget the label!
The Solution: Zend_Form

Create elements: Password:

```php
$password = new Zend_Form_Element_Text('password');
$password->addFilter('StringTrim')
    ->addValidator('StringLength', false, array(6))
    ->setRequired(true)
    ->setLabel('Password');
```

- Single filter
- Single validator
- Required
- Don't forget the label!
Create elements: the Login button:

```php
$login = new Zend_Form_Element_Submit('login');
$login->setRequired(false)
    ->setIgnore(true)
    ->setLabel('Login!');
```

- Need to display the button
- But we don't want to validate it or include it when pulling values
The Solution: Zend_Form

Create the Form object:

```php
// In a controller action:
$form = new Zend_Form();
$form->addElements(array($username, $password, $login));
if ($this->getRequest()->isPost()) {
    if ($form->isValid($this->getPost())) {
        // success
    }
}
$this->view->form = $form;
```

- Attach elements
- Check if valid – does all input filtering
- Pass it to the view
Create the view script:

• *Much* shorter!

```php
<h2>Please Login</h2>
<? $this->form ?>
The Solution: Zend_Form

First time viewing the form:

• Same as before!
The Solution: Zend_Form

Results when submitting empty values:

- Note: *required* flag has a correlation with the errors reported
The Solution: Zend_Form

Results when submitting invalid values:

- Note: errors are reported!
Zend_Form Features and Benefits

- Internationalization: localize your forms for your customers!
- Partial and Full Set data validation
- Filter and Validation chains per element
- Fully customizable output
- Adheres to Zend_Validate_Interface
  - Allows you to plug forms and/or elements in as validators for your models -- which means you could potentially replace Zend_Filter_Input classes in your models and thus make your models directly renderable!
- Break forms into visual and/or logical groups
Overview of Zend_Form's Architecture
Architecture Overview

• Base classes
  - forms
  - elements
  - display groups
  - sub forms

• Plugins
  - filters
  - validators
  - decorators
  - elements

• Utilities
  - plugin loaders
  - translators
Classes: Zend_Form

• **Model Forms**
  - Store and manipulate collections of elements and groups of elements
  - Validate attached elements and sub forms
  - Store and manipulate decorators for rendering the form

• **Class: Zend_Form**
Classes: Zend_Form_Element

- Store and manipulate element metadata
- Store and manipulate validator chains
- Store and manipulate filter chains
- Store and manipulate decorators for rendering element
- Base class: Zend_Form_Element

Element types:
- Button
- Checkbox
- Hash (CSRF protection)
- Hidden
- Image
- MultiCheckbox
- Multiselect
- Password
- Radio
- Reset
- Select
- Submit
- Text
- Textarea
Classes: Zend_Form_DisplayGroup

- Group elements visually when rendering
- Collection of one or more elements
- Order display group in form, and elements within display group
- Class: Zend_Form_DisplayGroup
• **Group elements logically**
  - For display purposes
  - For validation purposes

• **Potential uses**
  - Multi-page forms (each sub form used per page)
  - Dynamic forms (e.g., todo list, where each todo item is its own mini-form)

• **Class: Zend_Form_SubForm**
Plugins

• Utilizes Zend_Loader_PluginLoader for loading plugin classes
• Specify alternate class prefixes and paths to load:
  ▪ new plugins
  ▪ alternate versions of standard plugins
• Powerful and easy way to extend Zend_Form functionality
Plugins: Filters

- Normalize or filter input prior to validation
- Uses Zend_Filter classes by default

- Some available filters:
  - Alnum
  - Alpha
  - Digits
  - HtmlEntities
  - StringToLower
  - StringToUpper
  - StringTrim
  - StripTags
Plugins: Validators

- Validate input against one or more rules
- Uses Zend_Validate classes by default

- Some available validators:
  - Alnum
  - Alpha
  - Date
  - EmailAddress
  - InArray
  - Int
  - Regex
  - StringLength
Plugins: Decorators

- Render elements and forms by decorating them
- Uses pseudo-Decorator pattern
- More later...

- Some available decorators:
  - Callback
  - Description
  - Errors
  - Fieldset
  - Form
  - HtmlTag
  - Label
  - ViewHelper
  - ViewScript
Plugins: Elements

- Elements are loaded as plugins in Zend_Form
- You can create your own versions of standard elements, and still utilize Zend_Form's element factory methods

Some standard elements:
- Button
- Checkbox
- Password
- Radio
- Select
- Submit
- Text


Utilities

• **Plugin Loaders**
  - Load plugins
  - Register class prefixes and paths

• **Translators**
  - Zend_Translate and its adapters
  - Translate error messages and other translatable items
Zend_Form In-Depth
In-Depth: Plugins

• As noted, uses Zend_Loader_PluginLoader for loading plugins

• Resources considered plugins:
  ▪ Filters (elements only)
  ▪ Validators (elements only)
  ▪ Decorators (all types)
  ▪ Elements (forms only)

• Generally, specify a class prefix, path, and plugin type

• Allows specifying both new plugins as well as local replacements of standard plugins
In-Depth: Plugins

Example: Validator plugin

```php
// Specify additional or alternate validators for an element:
$element->addPrefixPath('My_Verify', 'My/Verify/', 'Verify');

class My_Verify_PASSWORD { /* ... */ };

// Load My_Verify_PASSWORD:
$element->addValidator('Password');
```
In-Depth: Plugins

Example: Decorator plugin

```php
// Specify additional or alternate decorators for the form:
$form->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');

class My_Decorator_FormElements { /* ... */ }

// Load FormElements decorator:
$form->addDecorator('FormElements');

// Outputs "My_Decorator_FormElements"
var_dump(get_class($form->getDecorator('FormElements')));
```

- **Replaces** standard “FormElements” decorator
In-Depth: Decorators

- Used to render elements, forms and groups
- Similar to *Decorator* pattern, but decorates string content using element and form metadata
- Each decorator decorates the content passed to it
  - Initial content is always an empty string
  - Return value *replaces* previous value
  - Decorator internally can append, prepend, or replace provided
  - Typically Stack decorators from inside -> out to create output
Example Decorator usage:

```php
$element->addDecorators(array(
    'ViewHelper',
    'Errors',
    'Description',
    array('HtmlTag', array('tag' => 'dd')),
    array('Label', array('tag' => 'dt')),
));
```

- ViewHelper to render element
- Element error messages (appends)
- Element hint (appends)
- Wrap in `<dd>`
- Element label in `<dt>` tag (prepends)
In-Depth: Decorators

- Each decorator has awareness of the element/form/etc.
  - Allows inspecting item to get metadata
  - Agnostic of class being decorated; retrieve with `getElement()` accessor regardless of class
  - Useful for building decorators that render only one aspect of an item
    - Label
    - Description
    - Errors
Some Standard Decorators

- **Callback**
  - Delegate to a specified callback

- **Description**
  - render from getDescription()

- **Errors**
  - render from getMessages()

- **Fieldset**
  - render content in a fieldset, with optional legend

- **FormElements**
  - Iterate through all elements, groups, and sub forms to generate content

- **Form**
  - Wrap content in an HTML form

- **HtmlTag**
  - Wrap content in HTML tags (or emit start or end tags)
In-Depth: Decorators

Some Standard Decorators

• **Image**
  - Render a form image

• **Label**
  - Render from `getLabel()`

• **ViewHelper**
  - Render using a view helper (typically pulled from element's 'helper' attribute)

• **ViewScript**
  - Render using a specified view script
Overview: What makes up an element?

• Metadata
• Filters
• Validators
• Decorators
Metadata

• Stored as “properties” of the element, via overloading
• Anything that can better qualify an element
• Typically for rendering
• Examples:
  ▪ CSS class(es)
  ▪ Javascript events (onClick, etc)
  ▪ Explicit XHTML id
  ▪ Javascript attribute hints (e.g., for Dojo)
Filters

• For normalizing input prior to validation
• Objects implementing Zend_Filter_Interface
• Attaching to elements:
  ▪ Instantiate and attach
  ▪ use addFilter(), addFilters(), or setFilters() as factories, using just the “short name” of the filter (class name minus common prefix)
• Use Zend_Form::setElementFilters() to set common filters for all elements en masse
Validators

• For validating input, to ensure it meets acceptance criteria

• Objects implementing Zend.Validate_Interface

• Attaching to elements:
  ▪ Instantiate and attach
  ▪ use addValidator(), addValidators(), or setValidators() as factories, using just the “short name” of the validator (class name minus common prefix)
Decorators

• For rendering as a form element
• Attaching to elements:
  ▪ Instantiate and attach
  ▪ use addDecorator(), addDecorators(), or setDecorators() as factories, using just the “short name” of the decorator (class name minus common prefix)
Decorators

- Default decorators for most elements:
  - ViewHelper
  - Errors
  - HtmlTag (<dd>)
  - Label (rendered in a <dt> tag)

- Some differ:
  - Submit, Reset, Button
  - Image

- Can set all elements en masse using Zend_Form::setElementDecorators()
Overview: What makes up a form?

• Metadata
• Elements
• Display Groups
• Sub Forms
• Ordering
• Decorators
• Validation Methods
Metadata

- Stored as “attrs” in the form
- Used to further qualify a form
  - action
  - method
  - CSS class
  - XHTML id
  - Javascript events
- Use various *Attrib(s)() accessor(s) to manipulate
Elements

• Can attach concrete instances, or create new ones

• `createElement()`
  - Create and return a new element, using paths and defaults set in the form object

• `addElement()/addElements/setElements()`
  - Create and attach one or more elements to the form
Display Groups

• Used to group elements visually when rendering
• Uses fieldset by default to group elements
• Use addDisplayGroup() as a factory to:
  ▪ create a new display group
  ▪ attach specified elements to a display group
  ▪ provide options specifying how the display group is rendered
    • decorators
    • legend
Sub Forms

• What are they used for?
  ▪ Way to group items logically
  ▪ Items that are related semantically
  ▪ Repeatable groups of items (e.g., todo list tasks)
  ▪ Single page of a multi-view form

• Extends Zend_Form
  ▪ Has all functionality of Zend_Form

• Use addSubForm() to add sub forms
  ▪ Accepts only concrete instances
In-Depth: Forms

Sub Forms

• Utilize Array Notation:

```html
<dt><label for="user-username" class="required">Username:</label></dt>
<dd><input name="user[username]" id="user-username" value="" type="text"></dd>

<dt><label for="user-password" class="required">Password:</label></dt>
<dd><input name="user[password]" id="user-password" value="" type="password"></dd>

<dt></dt>
<dd><input name="user[login]" id="user-login" value="Login!" type="submit"></dd>
```

• Uses sub form's name as array name
• Each element is a key in the array
• XHTML ids are inflected to use valid formats
• Array submitted for sub form is passed to sub form for validation
Ordering

- Each attached item – elements, display groups, sub forms – have their own order attribute; form object uses these to determine order
- Ordering performed only when iterating over form, rendering, or validating
- Elements attached to display groups
  - When rendering, element order honored by display group; *not* double-rendered (removed from overall order).
  - When validating, order is honored; display groups are ignored.
Decorators

- Default decorators for Zend_Form:
  - FormElements (iterates over elements, display groups, and sub forms)
  - HtmlTag (dl tag with class of 'zend_form')
  - Form (wrap content in XHTML form tag)
Decorators

- Default decorators for Zend_Form_SubForm and Zend_Form_DisplayGroup:
  - FormElements
  - HtmlTag (<dl> tag)
  - Fieldset (using legend for item, if available)
  - DtDdWrapper (to keep flow of parent form; empty <dt>, and remainder of content in <dd>
Validation Methods

• `isValid()`: validate entire form (except optional fields which are missing or empty)
• `isValidPartial()`: validate only fields that were submitted
• `getErrors()`: returns array of element => error codes
• `getMessages()`: returns array of element => error messages array
How it works:

- Create a Zend_Translate object
- Attach it to:
  - Zend_Validate_Abstract
    - setDefaultTranslator()
  - Zend_Form
    - setDefaultTranslator()
    - setTranslator()
  - Zend_Form_Element
    - setTranslator()
  - Or in Zend_Registry as key “Zend_Translate”
Standard Translatable Items:
• Validation error messages
• Labels
• Fieldset Legends
• Form and Element Descriptions
• Multi-option Values
• Submit and Button Labels
Using Forms in MVC Applications
Make your form reusable

• Extend Zend_Form
• In the constructor:
  ▪ Setup the form environment
  ▪ Initialize your elements
Forms in MVC Applications

Example:

class LoginForm extends Zend_Form
{
    public function __construct($options = null)
    {
        parent::__construct($options);
        $username = $this->createElement('text', 'username');
        $password = $this->createElement('password', 'password');
        $login = $this->createElement('submit', 'login');

        // Configure elements per original examples...

        // Add elements to form:
        $this->setElements(array($username, $password, $login));
    }
}
Integrate the form in your Controller

- Create a “landing page” for the form
  - Instantiate the form object
  - Pass it to the view object
  - Create a view script that renders the form object
Integrate the form in your Controller

- **Accessor for creating / retrieving the form object**
- “landing” page (index action) simply assigns the form to the view

```php
class LoginController extends Zend_Controller_Action
{
    protected $_form;

    public function getForm()
    {
        if (null === $this->_form) {
            $this->_form = new LoginForm(array(
                'action' => '/login/process',
                'method' => 'post',
            ));
        }
        return $this->_form;
    }

    public function indexAction()
    {
        $this->view->form = $this->getForm();
    }
}
```
Integrate the form in your Controller

• View script for landing page

```php
<h2>Please Login</h2>
<? $this->form ?>
```
Integrate the form in your Controller
• Create an action to process the form
  ▪ Instantiate the form object
  ▪ Pass submitted data to the form's isValid() method
    • If not valid, re-render the form
    • Use the action corresponding to the landing page action
  ▪ If valid, do something with it...
    • We'll redirect
Integrate the form in your Controller

- Check if we received a POST request
- If so, check if the form considers the submitted data valid
- Redirect after processing the data
- Redisplay the form on errors

```php
public function processAction()
{
    $request = $this->getRequest();
    if (!$request->isPost()) {
        return $this->_forward('index');
    }

    $form = $this->getForm();
    if ($form->isValid($request->getPost())) {
        // Pass to authentication object or whatnot...
        // And then redirect to success page
        return $this->_helper->redirector('success');
    }

    // Failed
    $this->view->form = $form;
    $this->render('index');
}
```
Wrapup
Topics not Covered

• Iteration of forms, groups, and sub forms
• Accessing element metadata
• Specifics of individual elements
• Specifics of individual decorators
• View helpers available and used by Zend_Form
• Specifics on translating validation messages and other items
• Configuring forms and elements using Zend_Config
• Much, much more!
Thank you!

More on Zend Framework:
http://framework.zend.com