Capítulo 15. Zend_Form

Traduzido por Flávio Gomes da Silva Lisboa

Sumário

15.1. Zend_Form

Zend_Form simplifica a criação e manipulação de formulários em sua aplicação web. Ele realiza os seguintes objetivos:

- Validação e filtragrem de elementos de entrada
- Ordenação de elementos
- Renderização de elementos e formulários, incluindo escaping
- Agrupamento de elementos e formulários
- Configuração no nível de formulários e elementos

Isso influencia fortemente outros componentes do Zend Framework a cumprir seus objetivos,

incluindo Zend_Config, Zend_Validate, Zend_Filter, Zend_Loader_PluginLoader, e opcionalmente Zend_View.

15.2. Zend_Form Quick Start

Esse guia rápido pretende cobrir o básico de criação, validação e renderização de formulários usando Zend_Form.

15.2.1. Criar um objeto formulário

A criação de um objeto formulário é muito simples: simplesmente instancie Zend_Form:

```
<?php
$form = new Zend_Form;
?>
```

Para casos de uso mais avançados, você pode querer criar uma subclasse de Zend_Form, mas para formulários simples, você pode criar um formulário programaticamente usando um objeto Zend_Form.

Se você deseja especificar os atributos action e method do formulário (sempre boas idéias), você pode fazê-lo com os acessores setAction() e setMethod():

```
<?php
$form->setAction('/resource/process')
     ->setMethod('post');
?>
```

O código acima configura o atributo action do formulário para a URL parcial "/resource/process" e o atributo method para HTTP POST. Isso será refletido durante a renderização final.

Você pode configurar atributos HTML adicionais para a tag <form> usando os métodos setAttrib() ou setAttribs(). Para a instância, se você deseja configurar o atributo “id”:

```
<?php
$form->setAttrib('id', 'login');
?>
```

15.2.2. Adicionando elementos ao formulário

Um formulário nada é sem seus elementos. Zend_Form vem embarcado com alguns elementos padrão que renderizam XHTML via Zend_View helpers. São os seguintes:

- button
- checkbox (ou vários checkboxes de uma vez com multiCheckbox)
- hidden
- image

- password
- radio
- reset
- select (tanto regular quanto multi-select)
- submit
- text
- textarea

Você tem duas opções para adicionar elementos a um formulário: você passar instâncias de elementos concretos, ou pode passar simplesmente o tipo de elemento e deixar `Zend_Form` instanciar um objeto do tipo correto para você.

Alguns exemplos:

```
<?php
// Instanciando um elemento e passando para o objeto form:
$form->addElement(new Zend_Form_Element_Text('username'));
// Passando um tipo de elemento de formulário para o objeto form:
$form->addElement('text', 'username');
?>
```

Por padrão, esses elementos não tem quaisquer validadores ou filtros. Isso significa que você precisa configurar seus elementos com, minimamente, validadores e, potencialmente, filtros. Você pode fazer isso (a) ou antes de passar o elemento para o formulário, (b) via opções de configuração passadas quando criamos um elemento via `Zend_Form`, ou (c) pela extração do elemento do objeto formulário e configuração do mesmo depois do fato.

Primeiro dê uma olhada na criação de validadores para uma instância de elemento concreta. Você pode passar objetos `Zend_Validate_*` , ou o nome de um validador:

```
<?php
$username = new Zend_Form_Element_Text('username');
// Passando um objeto Zend_Validate_*:
$username->addValidator(new Zend_Validate_Alnum());
// Passando um nome de validador:
$username->addValidator('alnum');
?>
```

Quando usamos essa segunda opção, se o validador pode aceitar argumentos do construtor, você pode passá-los em uma matriz como o terceiro parâmetro:

```
<?php
// Passa um padrão
$username->addValidator('regex', false, array('/^[a-z]/i'));
?>
```

(O segundo parâmetro é usado para indicar se há ou não falha desse validador em prevenir os validadores anteriores de rodar; por padrão, isso é falso.

Você pode também desejar especificar um elemento como obrigatório. Isso pode ser feito sem usar um acessor ou passar adiante uma opção quando criar o elemento. No primeiro caso:

```
<?php
// Faz este elemento obrigatório:
$username->setRequired(true);
?>
```

Quando um elemento é obrigatório, um validador 'NotEmpty' é adicionado ao topo do conjunto de validadores, garantindo que o elemento tenha um valor quando requerido.

Filtros são registrados basicamente dos mesmos modos que os validadores. Para propósitos de ilustração, adicionemos um filtro para caixa baixa ao valor final:

```
<?php
$username->addFilter('StringtoLower');
?>
```

Assim, nossa configuração final para o elemento poderia parecer com isso:

```
<?php
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]/'))
         ->setRequired(true)
         ->addFilter('StringToLower');
// ou, mais compactamente:
$username->addValidators(array('alnum',
        array('regex', false, '/^[a-z]/i')
    ))
    ->setRequired(true)
    ->addFilters(array('StringToLower'));
?>
```

Simples como isso é, fazer isso para cada elemento comum em um formulário pode ser um pouco tedioso. Tentemos a opção (b) acima. Quando nós criamos um novo elemento usando `Zend_Form::addElement()` como um padrão factory, nós podemos opcionalmente passá-los nas opções de configuração. Isso pode incluir a utilização de validadores e filtros. Assim, para fazer tudo acima implicitamente, tente o seguinte:

```
<?php
$form->addElement('text', 'username', array(
    'validators' => array(
        'alnum',
        array('regex', false, '/^[a-z]/i')
    ),
    'required' => true,
    'filters'  => array('StringToLower'),
));
?>
```

Nota

Se você acha que está configurando elementos usando as mesmas opções em muitos locais, você pode querer considerar a criação e utilização de sua própria subclasse `Zend_Form_Element`; isso irá proporcionar economia de digitação.

15.2.3. Renderizar um formulário

Renderizar um formulário é simples. A maioria dos elementos usa um helper `Zend_View` para renderizar a si mesmo, assim precisa de um objeto view. Além disso, você tem duas opções: use o método de formulário render(), ou simplesmente use o comando echo.

```
<?php
// Chama explicitamente render(), e passa um objeto view opcional:
echo $form->render($view);
// Assume que um objeto view  tem sido previamente configurado com setView():
echo $form;
?>
```

Por padrão, `Zend_Form` e `Zend_Form_Element` tentarão usar o objeto view inicializado no `ViewRenderer`, o que significa que você não precisa configurar o view manualmente quando usar o MVC Zend Framework. Renderizar um formulário em um view script é ainda mais simples:

```
<?= $this->form ?>
```

Sob seu disfarce, `Zend_Form` usa "decorators" para executar renderização. Esses decorators podem substituir conteúdo, adicionar conteúdo, ou incluir conteúdo em seu início, e ter completa introspecção para o elemento passado a eles. Como resultado, você pode combinar múltiplos decorators para obter efeitos customizados. Por padrão, `Zend_Form_Element` normalmente combina quatro decorators para obter sua saída; a configuração parece com algo assim:

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

(Onde <HELPERNAME> é o nome de um view helper a ser utilizado, e varia de acordo com o elemento.)

O código acima cria saída como a seguinte:

```
<dt><label for="username" class="required">Username</dt>
<dd>
    <input type="text" name="username" value="123-abc" />
    <ul class="errors">
        <li>'123-abc' não tem somente caracteres alfabéticos e dígitos</li>
        <li>'123-abc' não casa com o padrão '/^[a-z]/i'</li>
```

```
    </ul>
</dd>
```

(Embora não com a mesma formatação.)

Você pode alterar os decorators usados por um elemento se você quiser ter diferentes saídas; veja a seção sobre decorators para mais informações.

O formulário por si próprio simplesmente itera através dos elementos, e incorpora-os dentro de um `<form>` HTML. Os atributos action e method que você forneceu quando configurou o formulário são fornecidos para a tag `<form>`, como quaisquer atributos que você configure via `setAttribs()` e família.

Elementos são iterados ou na ordem na qual foram registrados, ou, se seu elemento contém um atributo de ordem, essa ordem será usada. Você pode configurar a ordem de um elemento usando:

```php
<?php
$element->setOrder(10);
?>
```

Ou, quando criar um elemento, pela passagem da mesma como uma opção:

```php
<?php
$form->addElement('text', 'username', array('order' => 10));
?>
```

15.2.4. Verifique se um formulário é válido

Depois que um formulário é submetido, você precisa verificar se ele passa pelas validações. Cada elemento é confrontado com o dado fornecido; se uma chave que case com o nome do elemento não estiver presente, e o item está marcado como obrigatório, as validações serão feitas com um valor null.

De onde os dados vêm? Você pode usar `$_POST` or `$_GET`, ou qualquer outra fonte de dados que você possa ter à mão (requisições de web service, por exemplo):

```php
<?php
if ($form->isValid($_POST)) {
    // successo!
} else {
    // falha!
}
?>
```

Com requisições AJAX, você pode algumas vezes escapar com validação de elementos simples, ou grupos de elementos. `isValidPartial()` irá validar um formulário parcial. A exceção de `isValid()`, entretanto, se uma chave particular não estiver presente, ela não irá rodar validações para aquele elemento particular:

```php
<?php
if ($form->isValidPartial($_POST)) {
```

```
    // elementos apresentam todas as validações passadas
} else {
    // um ou mais elementos testados falharam nas validações
}
?>
```

Um método adicional, `processAjax()`, pode também ser usado para validação parcial de formulários. Ao contrário de `isValidPartial()`, ele retorna uma string no formato JSON- contendo mensagens de erro em caso de falha.

Assumindo que suas validações tenham passado, você pode agora buscar os valores filtrados:

```
<?php
$values = $form->getValues();
?>
```

Se você precisar de valores não filtrados em qualquer ponto, use:

```
<?php
$unfiltered = $form->getUnfilteredValues();
?>
```

15.2.5. Obtenha o status de erro

As validações doseu formulário falharam? Na maioria dos casos, você pode simplesmente renderizar o formulário novamente, e erros serão exibidos quando usamos os decorators padrão:

```
<?php
if (!$form->isValid($_POST)) {
    echo $form;
    // ou associa ao objeto view e renderiza um view...
    $this->view->form = $form;
    return $this->render('form');
}
?>
```

Se você quer inspecionar os erros, você tem dois métodos. `getErrors()` retorna uma matriz associativa de nomes/ códigos de elemento (onde o código é uma matriz de códigos de erro). `getMessages()` retorna uma matriz associativa de nomes / mensagens (onde mensagens é uma matriz associativa de código de erro/ pares de mensagens de erro). Se um dado elemento não tem quaisquer erros, não será incluído na matriz.

15.2.6. Colocando tudo junto

Construamos um simples formulário de login. Ele precisará representar os elementos:

- username
- password
- submit

Para nossos propósitos, assumiremos que um username válido deve ser alfanumérico somente, começando com uma letra, tendo um comprimento mínimo de 6, e máximo de 20 caracteres; os caracteres serão normalizados para caixa baixa. Passwords devem ter um mínimo de 6 caracteres. Nós simplesmente lançaremos o valor de submit quando o formulário for submetido, assim ele pode permanecer invalidado.

Nós usaremos o poder das opções de configuração `Zend_Form`'s para construir o formulário:

```
<?php
$form = new Zend_Form();
$form->setAction('/usr/login')
     ->setMethod('post');
// Cria e configura o elemento username:
$username = $form->createElement('text', 'username');
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]+/'))
         ->addValidator('stringLength', false, array(6, 20))
         ->setRequired(true)
         ->addFilter('StringToLower');
// Cria e configura o elemento password:
$password = $form->createElement('password', 'password');
$password->addValidator('StringLength', false, array(6))
         ->setRequired(true);
// Adiciona elementos ao formulário:
$form->addElement($username)
     ->addElement($password)
     // usa addElement() como um padrão factory para criar o botão 'Login':
     ->addElement('submit', 'login', array('label' => 'Login'));
?>
```

Em seguida, nós criaremos um controlador para manipular isso:

```
<?php
class UserController extends Zend_Controller_Action
{
    public function getForm()
    {
        // cria o formulário como acima
        return $form;
    }
    public function indexAction()
    {
        // renderiza user/form.phtml
        $this->view->form = $this->getForm();
        $this->render('form');
    }
    public function loginAction()
    {
        if (!$this->getRequest()->isPost()) {
            return $this->_forward('index');
        }
        $form = $this->getForm();
        if (!$form->isValid($_POST)) {
            // falha na validação; exibe o formulário novamente
            $this->form = $form;
            return $this->render('form');
        }
```

```
            $values = $form->getValues();
            // agora tenta e autentica....
        }
}
?>
```

E um view script para exibir o formulário:

```
<h2>Please login:</h2>
<?= $this->form ?>
```

Como você irá notar a partir do código do controlador, há mais trabalho a fazer: enquanto a submissão pode ser válida, você pode ainda precisar fazer alguma autenticação usando `Zend_Auth`, por exemplo.

15.2.7. Usando um objeto Zend_Config

Todas as classes `Zend_Form` são configuráveis usando `Zend_Config`; você pode passar um objeto `Zend_Config` para o construtor ou passá-lo via `setConfig()`. Dê uma olhada em como nós podemos criar o formulário usando um arquivo INI. Primeiro, siga as recomendações, e coloque nossas configurações dentro de seções que reflitam o local editado, e foque na seção 'development'. Em seguida, nós configuraremos uma seção para o controlador dado ('user'), e uma chave para o formulário ('login'):

```
[development]
; general form metainformation
user.login.action = "/user/login"
user.login.method = "post"
; username element
user.login.elements.username.type = "text"
user.login.elements.username.options.validators.alnum.validator = "alnum"
user.login.elements.username.options.validators.regex.validator = "regex"
user.login.elements.username.options.validators.regex.options.pattern = "/^[a-
z]/i"
user.login.elements.username.options.validators.strlen.validator = "StringLength
"
user.login.elements.username.options.validators.strlen.options.min = "6"
user.login.elements.username.options.validators.strlen.options.max = "20"
user.login.elements.username.options.required = true
user.login.elements.username.options.filters.lower.filter = "StringToLower"
; password element
user.login.elements.password.type = "password"
user.login.elements.password.options.validators.strlen.validator = "StringLength
"
user.login.elements.password.options.validators.strlen.options.min = "6"
user.login.elements.password.options.required = true
; submit element
user.login.elements.submit.type = "submit"
```

Você pode então passar isso para o construtor do formulário:

```
<?php
$config = new Zend_Config_Ini($configFile, 'development');
$form   = new Zend_Form($config->user->login);
```

```
?>
```

e o formulário inteiro será definido.

15.2.8. Conclusão

Com esse pequeno tutorial, você deve agora estar pronto para liberar o poder e a flexibilidade de `Zend_Form`. Leia mais para informações avançadas!

15.3. Criando Elementos de Formulário Usando Zend_Form_Element

Um formulário é feito de elementos, que tipicamente correspondem à entrada de formulário HTML. Zend_Form_Element encapsula elementos de formulário simples, com as seguintes áreas de responsabilidade:

- validação (o dado submetido é válido?)
    - captura de mensagem e códigos de erro de validação
- filtragem (como o elemento é escapado[1] ou normalizado antes de validar e/ou enviar para a saída?)
- renderização (como o elemento é exibido?)
- metadados e atributos (que informação além dessa qualifica o elemento?)

A classe base, `Zend_Form_Element`, tem padrões razoáveis para muitos casos, mas é melhor estender a classe para elementos comumente usados para propósitos especiais. Adicionalmente, Zend Framework vem embarcado com um número de elementos padrão XHTML; você pode ler sobre eles no capítulo Elementos Padrão.

15.3.1. Plugin Loaders

`Zend_Form_Element` faz uso de Zend_Loader_PluginLoader para permitir aos desenvolvedores especificar locais de validadores, filtros e decorators alternativos. Cada um tem seu próprio plugin loader associado com ele, e acessores genéricos são usados para recuperar e modificar cada um.

The following loader types are used with the various plugin loader methods: 'validate', 'filter', and 'decorator'. The type names are case insensitive.

Os métodos usados para interagir com plugin loaders são os seguintes:

- `setPluginLoader($loader, $type)`: `$loader` é o próprio objeto plugin loader, enquanto `$type` é um dos tipos especificados acima. Isso configura o plugin loader para o tipo dado para o objeto loader especificado recentemente.
- `getPluginLoader($type)`: recupera o plugin loader associado com `$type`.
- `addPrefixPath($prefix, $path, $type = null)`: adiciona uma associação prefixo/caminho ao loader especificado por `$type`. Se `$type` é null, ele tentará adicionar o caminho a todos os loaders, pelo acréscimo do prefixo "_Validate", "_Filter", e "_Decorator"; e incrementará o caminho com "Validate/", "Filter/", e "Decorator/". Se você

1 Na falta de outro termo... quer dizer que comandos na string definida serão ignorados

tem todos as suas classes de elemento de formulário extras debaixo de uma hierarquia comum, esse é um método conveniente para configurar o prefixo base para eles.

- `addPrefixPaths(array $spec)`: permite que você adicione muitos caminhos de uma vez a um ou mais plugin loaders. Ele espera que cada item da matriz seja uma matriz com as chaves 'path', 'prefix', e 'type'.

Validators, filters, e decorators customizados são um modo fácil de compartilhar funcionalidade entre formulários e encapsular funcionalidades customizadas.

Exemplo 15.1. Rótulo Customizado

Um caso de uso comum para plugins é fornecer substituições para classes padrão. Por exemplo, se você quiser fornecer uma implementação diferente do decorator 'Label' – por exemplo, para sempre adicionar dois pontos – você poderia criar seu próprio decorator 'Label' com seu próprio prefixo de classe, e então adicioná-lo ao seu prefixo de caminho.

Comecemos com um decorator Label customizado. Nós daremos o prefixo de classe "My_Decorator", e a própria classe estará no arquivo "My/Decorator/Label.php".

```
<?php
class My_Decorator_Label extends Zend_Form_Decorator_Abstract
{
    protected $_placement = 'PREPEND';
    public function render($content)
    {
        if (null === ($element = $this->getElement())) {
            return $content;
        }
        if (!method_exists($element, 'getLabel')) {
            return $content;
        }
        $label = $element->getLabel() . ':';
        if (null === ($view = $element->getView())) {
            return $this->renderLabel($content, $label);
        }
        $label = $view->formLabel($element->getName(), $label);
        return $this->renderLabel($content, $label);
    }
    public function renderLabel($content, $label)
    {
        $placement = $this->getPlacement();
        $separator = $this->getSeparator();
        switch ($placement) {
            case 'APPEND':
                return $content . $separator . $label;
            case 'PREPEND':
            default:
                return $label . $separator . $content;
        }
    }
}
```

Agora nós podemos dizer ao elemento para usar esse caminho de plugin quando procurar por decorators:

```
$element->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

Alternativamente, nós podemos fazer com que o nível do formulário garanta que todos os decorators utilizarão esse caminho:

```
$form->addElementPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

Com esse caminho adicionado, quando você adiciona um decorator, o caminho 'My/Decorator/' será procurado primeiro para ver se o decorator existe lá. Como resultado, 'My_Decorator_Label' será usado agora quando o decorator 'Label' for requisitado.

15.3.2. Filtros

É freqüentemente útil e/ou necessário executar alguma normalização sobre a entrada antes da validação – por exemplo, você pode despojar todo HTML, mas rodas suas validações sobre o que permanece para garantir que a submissão seja válida. Ou você pode querer eliminar espaços em branco envolvendo a entrada de modo que um validator StringLength não retornará um falso positivo. Essas operações podem ser executadas usando `Zend_Filter`, e `Zend_Form_Element` tem suporte para cadeias de filtro, permitindo a você especificar filtros múltiplos e seqüenciais para serem utilizados. A filtragem acontece tanto durante a validação como quando você recupera o valor do elemento via `getValue()`:

```
<?php
$filtered = $element->getValue();
?>
```

Filtros podem ser adicionados a cadeia de dois modos:

- passando uma instância de filtro concreta
- fornecendo um nome de filtro – ou um nome curto ou o nome da classe qualificada completamente

Vejamos alguns exemplos:

```
<?php
// Instância concreta de filter:
$element->addFilter(new Zend_Filter_Alnum());
// Nome da classe qualificada completamente:
$element->addFilter('Zend_Filter_Alnum');
// Nome curto do filtro:
$element->addFilter('Alnum');
$element->addFilter('alnum');
?>
```

Nomes curtos são tipicamente o nome do filtro menos o prefixo. No caso padrão, isso significa menos o prefixo 'Zend_Filter_'. Adicionalmente, a primeira letra não precisa ser maiúscula.

Usando Classes de Filtro Customizadas

Se você tem seu próprio conjunto de classes de filtro, você pode falar a `Zend_Form_Element` sobre elas usando `addPrefixPath()`. Por exemplo, se você tem validadores debaixo do prefixo 'My_Filter', você pode falar a `Zend_Form_Element` sobre

isso como segue:

```
<?php
$element->addPrefixPath('My_Filter', 'My/Filter/', 'filter');
?>
```

(Lembre que o terceiro argumento indica qual plugin loader na qual a ação será executada.)

Se em qualquer momento você precisar de um valor não filtrado, use o método `getUnfilteredValue()`:

```
<?php
$unfiltered = $element->getUnfilteredValue();
?>
```

Para mais informações sobre filtros, veja a documentação Zend_Filter.

Métodos associados com filtros incluem:

- `addFilter($nameOrFilter, array $options = null)`
- `addFilters(array $filters)`
- `setFilters(array $filters)` (sobrescreve todos os filtros)
- `getFilter($name)` (recupera um objeto filtro pelo nome)
- `getFilters()` (recupera todos os filtros)
- `removeFilter($name)` (remove filtros pelo nome)
- `clearFilters()` (remove todos os filtros)

15.3.3. Validadores

Se você concorda com o mantra da segurança de "filtre a entrada, escape[2] a saída," você irá querer validar ("filtre a entrada") sua entrada de formulário. Em `Zend_Form`, cada elemento inclui sua própria cadeia de validadores, consistindo de validadores `Zend_Validate_*`.

Validadores podem ser adicionados à cadeia de dois modos:

- passando uma instância de validador concreta
- fornecendo um nome de validador – ou um nome curto ou um nome de classe qualificado completamente

Veja alguns exemplos:

```
<?php
// Instância de validador concreta:
$element->addValidator(new Zend_Validate_Alnum());
// Nome de classe qualificado completamente:
$element->addValidator('Zend_Validate_Alnum');
// Nome de validador curto:
```

2 Anule a interpretação de comandos contidos no texto da saída

```
$element->addValidator('Alnum');
$element->addValidator('alnum');
?>
```

Nomes curtos são tipicamente o nome do validador menos o prefixo. No caso padrão, isso significará menos o prefixo 'Zend_Validate_'. Adicionalmente, a primeira letra não precisa ser maiúscula.

Usando Classes de Validadores Customizadas

Se você tem seu próprio conjunto de classes de validadores, você pode falar a `Zend_Form_Element` sobre eles usando `addPrefixPath()`. Por exemplo, se você tem validadores debaixo do prefixo 'My_Validator', você pode falar a `Zend_Form_Element` sobre isso como segue:

```
<?php
$element->addPrefixPath('My_Validator', 'My/Validator/', 'validate');
?>
```

(Lembre que o terceiro argumento indica qual plugin loader executará qual ação)

Se a falha de uma validação particular deve prevenir validadores posteriores de incêndio, passe o booleano `true` como segundo parâmetro:

```
<?php
$element->addValidator('alnum', true);
?>
```

Se você quer usar um nome de string para adicionar um validador, e a classe do validador aceita argumentos para o construtor, você pode passá-los para o terceiro parâmetro de `addValidator()` como uma matriz:

```
<?php
$element->addValidator('StringLength', false, array(6, 20));
?>
```

Argumentos passados desse modo deverão estar na ordem na qual eles são definidos no construtor. O exemplo acima instanciará a classe `Zend_Validate_StringLenth` com seus parâmetros `$min` e `$max`:

```
<?php
$validator = new Zend_Validate_StringLength(6, 20);
?>
```

Fornecendo Mensagens de Erro de Validador Customizadas

Alguns desenvolvedores podem querer fornecer mensagens de erro customizadas para um validador. O argumento `$options` de `Zend_Form_Element::addValidator()`

permitem a você fazer isso pelo provimento da chave 'messages' e configurá-la para uma matriz de pares chave/valor para configurar os templates de mensagem. Você precisará saber os códigos de erro de vários tipos de erro de validação para o validador particular.

Uma opção melhor é usar um `Zend_Translate_Adapter` com seu formulário. Códigos de erro são automaticamente passados para o adaptador pelo decorator padrão Errors; você pode então especificar suas próprias strings de mensagem de erro pela configuração de traduções para os vários códigos de erro de seus validadores.

Você pode também configurar muitos validadores de uma vez, usando `addValidators()`. O uso básico é passar uma matriz de matrizes, com cada matriz contendo de 1 a 3 valores, casando com o construtor de `addValidator()`:

```
<?php
$element->addValidators(array(
    array('NotEmpty', true),
    array('alnum'),
    array('stringLength', false, array(6, 20)),
));
?>
```

Se você quer ser mais prolixo ou explícito, você pode usar as chaves de matriz 'validator', 'breakChainOnFailure', e 'options':

```
<?php
$element->addValidators(array(
    array(
        'validator'           => 'NotEmpty',
        'breakChainOnFailure' => true),
    array('validator' => 'alnum'),
    array(
        'validator' => 'stringLength',
        'options'   => array(6, 20)),
));
?>
```

Esse uso é bom para ilustrar como você poderia então configurar validadores em um arquivo de configuração:

```
element.validators.notempty.validator = "NotEmpty"
element.validators.notempty.breakChainOnFailure = true
element.validators.alnum.validator = "Alnum"
element.validators.strlen.validator = "StringLength"
element.validators.strlen.options.min = 6
element.validators.strlen.options.max = 20
```

Note que cada item tem uma chave, se precisa ou não; isso é uma limitação de usar arquivos de configuração – mas também ajuda a tornar explícito para que os argumentos são. Apenas lembre que quaisquer opções de validador devem ser especificadas na ordem.

Para validar um elemento, passe o valor para `isValid()`:

```php
<?php
if ($element->isValid($value)) {
    // válido
} else {
    // inválido
}
?>
```

Validação Opera Sobre Valores Filtrados

`Zend_Form_Element::isValid()` filtra valores por meio da cadeia de filtro fornecida antes da validação. Veja a seção Filters para mais informações.

Contexto de Validação

`Zend_Form_Element::isValid()` suporta um argumento adicional, `$context`. `Zend_Form::isValid()` passa a matriz inteira de dados sendo processados para `$context` quando validar um formulário, e `Zend_Form_Element::isValid()`, por sua vez, passa-a para cada validador. Isso significa que você pode escrever validadores que estão atentos aos dados passados para outros elementos de formulário. Como um exemplo, considere um formulário de registro padrão que tem campos tanto para a senha quanto para a confirmação de senha; uma validação deeria ser que os dois campos casem. Tal validador poderia parecer com o seguinte:

```php
<?php
class My_Validate_PasswordConfirmation extends Zend_Validate_Abstract
{
    const NOT_MATCH = 'notMatch';
    protected $_messageTemplates = array(
        self::NOT_MATCH => 'Password confirmation does not match'
    );
    public function isValid($value, $context = null)
    {
        $value = (string) $value;
        $this->_setValue($value);
        if (is_array($context)) {
            if (isset($context['password_confirm'])
                && ($value == $context['password_confirm']))
            {
                return true;
            }
        } elseif (is_string($context) && ($value == $context)) {
            return true;
        }
        $this->_error(self::NOT_MATCH);
        return false;
    }
}
?>
```

Validadores são processados na ordem. Cada validador é processado, exceto um validador criado com um valor Validators are processed in order. Cada validador é processado, a menos que um validador criado com um valor `breakChainOnFailure` igual a true falhe sua validação. Certifique-se de especificar suas validações em uma ordem razoável.

Depois de uma validação falha, você pode recuperar as mensagens e códigos de erro da cadeia do validador:

```
<?php
$errors   = $element->getErrors();
$messages = $element->getMessages();
?>
```

(Nota: mensagens de erro retornadas são uma matriz associativa de pares de mensagem código / erro.)

Em adição aos validadores, você pode especificar que um elemento é obrigatório, usando `setRequired(true)`. Por padrão, esse marco é false, significando que sua cadeia de validador será pulada se nenhum valor for passado para `isValid()`. Você pode modificar esse comportamento de vários modos:

- Por padrão, quando um elemento é obrigatório, um marco, 'allowEmpty', também é true. Isso significa que se um valor avaliado como vazio é passado para `isValid()`, os validadores serão pulados. Você pode chavear esse marco usando o acessor `setAllowEmpty($flag)`; quando o marco é false, então um valor é passado, os validadores ainda rodarão.
- Por padrão, se um elemento é obrigatório, mas não contém um validador 'NotEmpty', `isValid()` adicionará um no topo da pilha, com o marco `breakChainOnFailure` configurado. Isso faz com que o marco obrigatório tenha significado semântico: se nenhum valor for passado, nós imediatamente invalidamos a submissão e notificamos o usuário, e prevenimos outros validadores de rodar sobre o que nós realmente sabemos serem dados inválidos.

  Se você não quer esse comportamento, você pode desligá-lo passando um valor false para `setAutoInsertNotEmptyValidator($flag)`; isso irá prevenir `isValid()` de colocar o validador 'NotEmpty' na cadeia de validação.

Para mais informação sobre validadores, veja a documentação Zend_Validate.

Usando Zend_Form_Elements como validadores de propósito geral

`Zend_Form_Element` implementa `Zend_Validate_Interface`, significando que um elemento pode também ser usado como um validador em outras cadeias de validação relacionadas a não formulários.

Métodos associados com validação incluem:

- `setRequired($flag)` e `getRequired()` permite a você configurar e recuperar o marco 'requerido'. Quando configurar para o booleano `true`, esse marco requer que os elemento esteja no dados processado por `Zend_Form`.
- `setAllowEmpty($flag)` e `getAllowEmpty()` permite a você modificar o comportamento de elementos opcionais (por exemplo, elementos onde o marco requerido é false). Quando o marco 'allow empty' é true, valores vazios não serão passados para a cadeia de validador.
- `setAutoInsertNotEmptyValidator($flag)` permite a você especificar se um validador 'NotEmpty' irá ou não ser adicionado ao início de uma cadeia de validador quando o elemento é obrigatório. Por padrão, este marco é true.

- `addValidator($nameOrValidator, $breakChainOnFailure = false, array $options = null)`
- `addValidators(array $validators)`
- `setValidators(array $validators)` (sobrescreve todos os validadores)
- `getValidator($name)` (recupera um objeto validador pelo nome)
- `getValidators()` (recupera todos os validadores)
- `removeValidator($name)` (remove o validador pelo nome)
- `clearValidators()` (remove todos os validadores)

15.3.4. Decorators

Um ponto dolorido particular para muitos desenvolvedores web é a criação de formulários XHTML por eles mesmos. Para cada elemento, o desenvolvedor precisa criar marcação para o próprio elemento, tipicamente um rótulo, e, se eles estão sendo gentis para seus usuários, marcações para exibir mensagens de erro de validação. Quanto mais elementos na página, menos trivial essa tarefa se torna.

`Zend_Form_Element` tenta resolver esse problema por meio do uso de "decorators". Decorators são simplesmente classes que tem acesso ao elemento e um método para renderizar conteúdo. Para mais informações sobre como os decorators funcionam, por favor veja a seção sobre Zend_Form_Decorator.

Os decorators padrão usados por `Zend_Form_Element` são:

- *ViewHelper*: especifica um view helper para ser usado para renderizar o elemento. O atributo de elemento 'helper' pode ser usado para especificar qual view helper será usado. Por padrão, `Zend_Form_Element` especifica o view helper 'formText', mas subclasses individuais especificam helpers diferentes.
- *Errors*: adiciona mensagens de erro ao elemento usando `Zend_View_Helper_FormErrors`. Se nenhum está presente, nada é adicionado.
- *HtmlTag*: encobre o elemento e erros em uma tag HTML <dd>.
- *Label*: adiciona no início de um rótulo o elemento usando `Zend_View_Helper_FormLabel`, e envolve-o em uma tag <dt>. Se nenhum rótulo for fornecido, apenas a tag de definição de termo é renderizada.

Decorators Padrão Não Precisam Ser Carregados

Por padrão, os decorators padrão são carregados durante a inicialização do objeto. Você pode desabilitar isso passando a opção 'disableLoadDefaultDecorators' para o construtor:

```
<?php
$element = new Zend_Form_Element('foo', array('disableLoadDefaultDecorators
' => true));
```

Essa opção podem ser misturada com quaisquer outras opções que você passar, ambas como opções de matriz ou em um objeto `Zend_Config`.

Uma vez que a ordem na qual os decorators são registrados importa – o primeiro decorator a ser registrado é executado primeiro – você precisará certificar-se de registrar seus decorators em uma ordem apropriada, ou garantir que você configure as opções de colocação em uma ordem sensata. Para dar um exemplo, aqui está um código que registra os decorators padrão:

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

O contexto inicial é criado pelo decorator 'ViewHelper' que cria o próprio elemento do formulário. Em seguida, o decorator 'Errors' busca as mensagens de erro do elemento, e, se qualquer um estiver presente, passa-os para o view helper 'FormErrors' para renderizar. O próximo decorator, 'HtmlTag', envolve o elemento e erros em uma tag HTML <dd>. Finalmente, o último decorator, 'label', recupera o rótulo do elemento e passa para o view helper 'FormLabel', envolvendo-o em uma tag HTML <dt>; o valor é adicionado ao início do conteúdo por padrão. A saída resultante parece basicamente com isso:

```
<dt><label for="foo" class="optional">Foo</label></dt>
<dd>
    <input type="text" name="foo" id="foo" value="123" />
    <ul class="errors">
        <li>"123" is not an alphanumeric value</li>
    </ul>
</dd>
```

Para mais informações sobre decorators, leia a seção Zend_Form_Decorator.

Usando Decorators Múltiplos do Mesmo Tipo

Internamente, `Zend_Form_Element` usa uma classe de decorator como o mecanismo de busca quando recupera decorators. Como um resultado, você não pode registrar múltiplos decorators do mesmo tipo; decorators subseqüentes simplesmente sobrescreverão os que existirem antes.

Para contornar isso, você pode usar *aliases*. Ao invés de passar um decorator ou nome de decorator como primeiro argumento para `addDecorator()`, passe uma matriz com um elemento simples, com o alias apontando para o objeto ou seu nome:

```
<?php
// Alias para 'FooBar':
$element->addDecorator(array('FooBar' => 'HtmlTag'), array('tag' => 'div
'));
// E recupera mais tarde:
$decorator = $element->getDecorator('FooBar');
?>
```

Nos métodos `addDecorators()` e `setDecorators()`, você necessitará passar a

opção 'decorator' na matriz representando o decorator:

```
<?php
// Adiciona dois decorators 'HtmlTag' , apelidando um como 'FooBar':
$element->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);
// E recupera mais tarde:
$htmlTag = $element->getDecorator('HtmlTag');
$fooBar  = $element->getDecorator('FooBar');
?>
```

Métodos associados com decorators incluem:

- `addDecorator($nameOrDecorator, array $options = null)`
- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)` (sobrescreve todos os decorators)
- `getDecorator($name)` (recuperar um objeto decorator pelo nome)
- `getDecorators()` (recupera todos os decorators)
- `removeDecorator($name)` (remove decorator pelo nome)
- `clearDecorators()` (remove todos os decorators)

15.3.5. Metadados e Atributos

`Zend_Form_Element` manipula uma variedade de atributos e metadados de elementos. Os atributos básicos incluem:

- *name*: o nome do elemento. Usa os acessores `setName()` e `getName()`.
- *label*: o elemento rótulo. Usa os acessores `setLabel()` e `getLabel()`.
- *order*: o índice em que um elemento deve aparecer no formulário. Usa os acessores `setOrder()` e `getOrder()`.
- *value*: o valor atual do elemento. Usa os acessores `setValue()` e `getValue()`.
- *description*: uma descrição do elemento; freqüentemente usado para fornecer um aviso ou dica contextual javascript descrevendo o propósito do elemento. Usa os acessores `setDescription()` e `getDescription()`.
- *required*: marco indicando se o elemento é obrigatóri oou não quando executar a validação do formulário. Usa os acessores `setRequired()` e `getRequired()`. Esse marco é false por padrão.
- *allowEmpty*: marco indicando se um elemento não-obrigatório (opcional) deve ou não tentar validar valores vazios. Quando true, e o marco required é false, valores vazios não são passados para a cadeia do validador, e presume-se true. Usa os acessores `setAllowEmpty()` e `getAllowEmpty()`. Esse marco é true por padrão.

- *autoInsertNotEmptyValidator*: marco indicando se insere ou não um validador 'NotEmpty' quando o elemento é obrigatório. Por padrão, esse marco é true. Configura o marco com `setAutoInsertNotEmptyValidator($flag)` e determina o valor com `autoInsertNotEmptyValidator()`.

Elementos do formulário podem requerer metadados adicionais. Para elementos de formulário XHTML, por exemplo, você pode querer especificar atributos tais como a classe ou id. Para facilitar isso há um conjunto de acessores:

- *setAttrib($name, $value)*: adiciona um atributo
- *addAttribs(array $attribs)*: adiciona muitos atributos de uma vez
- *setAttribs(array $attribs)*: como addAttribs(), mas sobrescreve
- *getAttrib($name)*: recupera um valor de atributo simples
- *getAttribs()*: recupera todos os atributos como pares chave/valor
- *removeAttrib($name)*: remove um atributo simples
- *clearAttribs()*: limpa todos os atributos

A maior parte das vezes, entretanto, você pode simplesmente acessá-los como propriedades de objeto, já que `Zend_Form_Element` utiliza sobrecarga para facilitar acesso a eles:

```
<?php
// Equivalente a $element->setAttrib('class', 'text'):
$element->class = text;
?>
```

Por padrão, todos os atributos são passados para o view helper usado pelo elemento durante a renderização como atributos HTML do elemento tag.

15.3.6. Elementos Padrão

`Zend_Form` vem embarcado com um número de elementos padrão; por favor leia o capítulo Elementos Padrão para detalhes finais.

15.3.7. Métodos Zend_Form_Element

`Zend_Form_Element` tem muitos, muitos métodos. O que segue é um rápido sumário de suas assinaturas, agrupadas por tipo:

- Configuração:
    - `setOptions(array $options)`
    - `setConfig(Zend_Config $config)`
- I18N:
    - `setTranslator(Zend_Translate_Adapter $translator = null)`
    - `getTranslator()`
    - `setDisableTranslator($flag)`
    - `translatorIsDisabled()`

- Propriedades:
  - `setName($name)`
  - `getName()`
  - `setValue($value)`
  - `getValue()`
  - `getUnfilteredValue()`
  - `setLabel($label)`
  - `getLabel()`
  - `setDescription($description)`
  - `getDescription()`
  - `setOrder($order)`
  - `getOrder()`
  - `setRequired($flag)`
  - `getRequired()`
  - `setAllowEmpty($flag)`
  - `getAllowEmpty()`
  - `setAutoInsertNotEmptyValidator($flag)`
  - `autoInsertNotEmptyValidator()`
  - `setIgnore($flag)`
  - `getIgnore()`
  - `getType()`
  - `setAttrib($name, $value)`
  - `setAttribs(array $attribs)`
  - `getAttrib($name)`
  - `getAttribs()`
- Caminhos e loaders de plugin loaders:
  - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type)`
  - `getPluginLoader($type)`
  - `addPrefixPath($prefix, $path, $type = null)`
  - `addPrefixPaths(array $spec)`
- Validação:
  - `addValidator($validator, $breakChainOnFailure = false, $options = array())`
  - `addValidators(array $validators)`
  - `setValidators(array $validators)`

    - `getValidator($name)`
    - `getValidators()`
    - `removeValidator($name)`
    - `clearValidators()`
    - `isValid($value, $context = null)`
    - `getErrors()`
    - `getMessages()`
- Filtros:
    - `addFilter($filter, $options = array())`
    - `addFilters(array $filters)`
    - `setFilters(array $filters)`
    - `getFilter($name)`
    - `getFilters()`
    - `removeFilter($name)`
    - `clearFilters()`
- Renderização:
    - `setView(Zend_View_Interface $view = null)`
    - `getView()`
    - `addDecorator($decorator, $options = null)`
    - `addDecorators(array $decorators)`
    - `setDecorators(array $decorators)`
    - `getDecorator($name)`
    - `getDecorators()`
    - `removeDecorator($name)`
    - `clearDecorators()`
    - `render(Zend_View_Interface $view = null)`

15.3.8. Configuração

O construtor `Zend_Form_Element` aceita ou uma matriz de opções ou um objeto `Zend_Config` contendo opções, e isso pode ser configurado usando tanto `setOptions()` quanto `setConfig()`. Geralmente falando, chaves são nomeadas como se segue:

- Se a chave 'set' + key refere-se a um método `Zend_Form_Element`, então o valor fornecido será passado para aquele método.
- Caso contrário, o valor será usado para configurar um atributo.

Exceções à regra incluem o seguinte:

- `prefixPath` será passado para `addPrefixPaths()`

- Os seguintes modificadores não podem ser configurados desse modo:
    - `setAttrib` (ainda que `setAttribs` funcione)
    - `setConfig`
    - `setOptions`
    - `setPluginLoader`
    - `setTranslator`
    - `setView`

Como um exemplo, aqui está um arquivo de configuração que passa configuração para cada tipo de dado de configuração:

```
[element]
name = "foo"
value = "foobar"
label = "Foo:"
order = 10
required = true
allowEmpty = false
autoInsertNotEmptyValidator = true
description = "Foo elements are for examples"
ignore = false
attribs.id = "foo"
attribs.class = "element"
onclick = "autoComplete(this, '/form/autocomplete/element')" ; sets 'onclick' at
tribute
prefixPaths.decorator.prefix = "My_Decorator"
prefixPaths.decorator.path = "My/Decorator/"
disableTranslator = 0
validators.required.validator = "NotEmpty"
validators.required.breakChainOnFailure = true
validators.alpha.validator = "alpha"
validators.regex.validator = "regex"
validators.regex.options.pattern = "/^[A-F].*/$"
filters.ucase.filter = "StringToUpper"
decorators.element.decorator = "ViewHelper"
decorators.element.options.helper = "FormText"
decorators.label.decorator = "Label"
```

15.3.9. Elementos Customizados

Você pode criar seus próprios elementos customizados pela simples extensão da classe `Zend_Form_Element`. Razões comuns para fazer isso incluem:

- Elementos que compartilham validadores e/ou filtros comuns
- Elementos que tem a funcionalidade decorator

Há dois métodos tipicamente usados para estender um elemento: `init()`, que pode ser ser usado para adicionar a lógica de inicialização customizada ao seu elemento, e `loadDefaultDecorators()`, que pode ser usada para configurar uma lista de decorators usada por seu elemento.

Como um exemplo, digamos que todos os elementos de texto em um formulário que você está criando precisa ser filtrado com `StringTrim`, validado com uma expressão regular comum, e que

você use um decorator que você tenha criado para os mostrá-los, 'My_Decorator_TextItem'; adicionalmente, você tem um número de atributos padrão, incluindo 'size', 'maxLength', e 'class'. Você deseja especificar . Você poderia definir tal elemento como segue:

```
<?php
class My_Element_Text extends Zend_Form_Element
{
    public function init()
    {
        $this->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator')
             ->addFilters('StringTrim')
             ->addValidator('Regex', false, array('/^[a-z0-9]{6,}$/i'))
             ->addDecorator('TextItem')
             ->setAttrib('size', 30)
             ->setAttrib('maxLength', 45)
             ->setAttrib('class', 'text');
    }
}
?>
```

Você poderia então informar seu objeto de formulário sobre o prefixo de caminho para tais elementos, e começar a criar elementos:

```
<?php
$form->addPrefixPath('My_Element', 'My/Element/', 'element')
     ->addElement('foo', 'text');
?>
```

O elemento 'foo' será agora do tipo `My_Element_Text`, e exibirá o comportamento que você esboçar.

Outro método que você pode querer sobrescrever quando estender `Zend_Form_Element` é o método `loadDefaultDecorators()`. Esse método condicionalmente carrega um conjunto de decorators padrão para seu elemento; você pode desejar substituir por seus próprios decorators em sua classe estendida:

```
<?php
class My_Element_Text extends Zend_Form_Element
{
    public function loadDefaultDecorators()
    {
        $this->addDecorator('ViewHelper')
             ->addDecorator('DisplayError')
             ->addDecorator('Label')
             ->addDecorator('HtmlTag', array('tag' => 'div', 'class' => 'element
'));
    }
}
?>
```

Há muitas formas de customizar elementos; esteja certo de ler a documentação da API de `Zend_Form_Element` para saber todos os métodos disponíveis.

15.4. Criando Formulários Usando Zend_Form

A classe `Zend_Form` é usada para agregar elementos de formulário, exibir grupos, e subformulários. Ele pode então executar as seguintes ações sobre esses itens:

- Validação, incluindo recuperação de códigos e mensagens de erro
- Agregação de valor, incluindo população de itens e recuperação tant ode valores filtrados quanto de não filtrados de todos os itens
- Iteração sobre todos os itens, na ordem em que eles entram ou baseado na ordem das dicas recuperadas de cada item
- Renderização do formulário inteiro, ou via um decorator simples que executa renderização customizada ou pela iteração sobre cada item no formulário

Enquanto formulários criados com `Zend_Form` podem ser complexos, provavelmente o melhor caso de uso é para formulários simples; seu melhor uso é para Desenvolvimento Rápido de Aplicações e prototipação.

No mais básico, você simplesmente instancia um objeto formulário:

```
<?php
// Objeto formulário genérico:
$form = new Zend_Form();
// Objeto formulário customizado:
$form = new My_Form()
?>
```

Você pode opcionalmente passar a configuração, que será usada para configurar o estado do objeto, assim como potencialmente criar novos elementos:

```
<?php
// Passar em opções de configuração:
$form = new Zend_Form($config);
?>
```

`Zend_Form` é iterável, e irá iterar sobre elementos, grupos de exibição, e subformulários, usando a ordem em que eles foram registrados e qualquer índice de ordem que cada um pode ter. Isso é útil em casos onde você deseja renderizar os elementos manualmente na ordem apropriada.

A mágica de `Zend_Form` reside em sua habilidade de servir como um padrão factory para elementos e grupos de exibição, assim como a habilidade de renderizar a si próprio através de decorators.

15.4.1. Plugin Loaders

`Zend_Form` faz uso de `Zend_Loader_PluginLoader` para permitir aos desenvolvedores especificar locações de elementos decorators alternados. Cada um tem seu próprio plugin loader associado com ele, e acessores gerais são usados para recuperar e modificar cada um.

Os seguintes tipos de loader são usados com os vários métodos plugin loader: 'element' e 'decorator'. Os nomes de tipo são case insensitive.

Os métodos usados para interagir com plugin loaders são como segue:

- `setPluginLoader($loader, $type)`: $loader é o próprio plugin loader, enquanto $type é um dos tipos especificados acima. Isso configura o plugin loader para o tipo dado para o objeto loader especificado recentemente.
- `getPluginLoader($type)`: recupera o plugin loader associado com $type.
- `addPrefixPath($prefix, $path, $type = null)`: adiciona uma associação prefixo/caminho ao loader especificado por $type. Se $type é null, ele tentará adicionar o caminho a todos os loaders, pela adição do prefixo de cada um com "_Element" e "_Decorator"; e adição do caminho com "Element/" e "Decorator/". Se você tem todos as classes de elemento do formulário extra debaixo de uma hierarquia comum, isso é um método conveniente para configurar o prefixo base para eles.
- `addPrefixPaths(array $spec)`: permite que você adicione muitos caminhos de uma vez a um ou mais plugin loaders. Ele espera que cada item de matriz seja uma matriz com as chaves 'path', 'prefix', e 'type'.

Adicionalmente, você pode especificar prefixos de caminhos para todos os elementos e grupos de exibição criados através de instância de `Zend_Form` usando os seguintes métodos:

- `addElementPrefixPath($prefix, $path, $type = null)`: assim como `addPrefixPath()`, você deve especificar um prefixo de classe e um caminho. `$type`, quando especificado, deve ser um dos tipos de plugin loader types especificados por `Zend_Form_Element`; veja a seção de plugins de elemento para mais informações sobre valores `$type` válidos. Se nenhum `$type` for especificado, o método assumirá que é um prefixo genérico para todos os tipos.
- `addDisplayGroupPrefixPath($prefix, $path)`: assim como `addPrefixPath()`, você deve especificar um prefixo de classe e um caminho; entretanto, uma vez que display groups suportam somente decorators como plugins, nenhum `$type` é necessário.

Elementos customizados e decorators são um modo fácil de compartilhar funcionalidades entre formulários e encapsular funcionalidades customizadas. Veja o exemplo de Rótulo Customizado na documentação de elementos para um exemplo de como elementos customizados podem ser utilizados como substituição para classes padrão.

15.4.2. Elementos

`Zend_Form` fornece vários acessores para adicionar e remover elementos do formulário. Eles podem tomar instâncias de objetos elementos ou servem como fábricas para instanciar os objetos elementos por si mesmos.

O método mais básico para adicionar um elemento é `addElement()`. Esse método pode tomar um objeto do tipo `Zend_Form_Element` (ou de uma classe que estenda `Zend_Form_Element`), ou argumentos para construir um novo elemento – incluindo o elemento tipo, nome e quaisquer opções de configuração.

Como alguns exemplos:

```
<?php
// Usando uma instância de elemento:
$element = new Zend_Form_Element_Text('foo');
$form->addElement($element);
// Usando um padrão factory
```

```
//
// Cria um elemento do tipo type Zend_Form_Element_Text com o
// nome de 'foo':
$form->addElement('text', 'foo');
// Passa opção de rótulo para o elemento:
$form->addElement('text', 'foo', array('label' => 'Foo:'));
?>
```

addElement() Implementa Interface Fluente

`addElement()` implementa uma interface fluente; quer dizer, ele retorna o objeto `Zend_Form`, e não o elemento. Isso é feito para permitir a você encadear juntos múltiplos métodos addElement() ou outros métodos de formulário que implementam a interface fluente (todos os modificadores em Zend_Form implementam o padrão).

Se ao invés disso você deseja retornar o elemento, use `createElement()`, que é exibido abaixo. Fique atento, entretanto, que `createElement()` não anexa o elemento ao formulário.

Internamente, `addElement()` normalmente usa `createElement()` para criar o elemento antes de anexá-lo ao formulário.

Uma vez que um elemento tenha sido adicionado ao formulário, você pode recuperá-lo pelo nome. Isso pode ser feito usando o método `getElement()` ou usando sobrecarga para acessar o elemento como uma propriedade do objeto:

```
<?php
// getElement():
$foo = $form->getElement('foo');
// Como uma propriedade do objeto:
$foo = $form->foo;
?>
```

Ocasionalmente, você pode querer criar um elemento sem anexá-lo ao formulário (por exemplo, se você desejar fazer uso de vários caminhos de plugin registrados com o formulário, mas deseja mais tarde anexar o objeto ao subformulário). O método `createElement()` permite a você fazer assim:

```
<?php
// $username torna-se um objeto Zend_Form_Element_Text:
$username = $form->createElement('text', 'username');
?>
```

15.4.2.1. Populando e Recuperando Valores

Depois de validar um formulário, você tipicamente precisará recuperar os valores de modo que você possa executar outras operações, tais como atualizar um banco de dados ou notificar um web service. Você pode recuperar todos os valores para todos os elementos usando `getValues()`; `getValue($name)` permite a você recuperar um valor de elemento simples pelo nome do elemento:

```php
<?php
// Pega todos os valores:
$values = $form->getValues();
// Pega somente o valor do elemento 'foo':
$value = $form->getValue('foo');
?>
```

Algumas vezes você irá querer popular o formulário com valores específicos antes de renderizar. Isso pode ser feito com os métodos `setDefaults()` ou `populate()`:

```php
<?php
$form->setDefaults($data);
$form->populate($data);
?>
```

15.4.2.2. Operações Globais

Ocasionalmente você irá querer certas operações para afetar todos os elementos. Cenários comuns incluem a necessidade de configurar caminhos de prefixo de plugin para todos os elementos, configurar decorators para todos os elementos, e configurar filtros para todos os elementos. Como exemplos:

Exemplo 15.2. Configurar Caminhos de Prefixo para Todos os Elementos

Você pode configurar caminhos de prefixo para todos os elementos por tipo, ou usando um prefixo global. Como exemplos:

```php
<?php
// Configura o caminho de prefixo global:
// Cria os caminhos para prefixos My_Foo_Filter, My_Foo_Validate,
// e My_Foo_Decorator
$form->addElementPrefixPath('My_Foo', 'My/Foo/');
// Apenas caminhos de filtro:
$form->addElementPrefixPath('My_Foo_Filter', 'My/Foo/Filter', 'filter');
// Apenas caminhos de validadores:
$form->addElementPrefixPath('My_Foo_Validate', 'My/Foo/Validate', 'validate');
// Apenas caminhos de decorator:
$form->addElementPrefixPath('My_Foo_Decorator', 'My/Foo/Decorator', 'decorator
');
?>
```

Exemplo 15.3. Configurar Decorators para Todos os Elementos

Você pode configurar decorators para todos os elementos. `setElementDecorators()` aceita uma matriz de decorators, exatamente como `setDecorators()`, e irá sobrescrever quaisquer decorators previamente configurados em cada elemento. Neste exemplo, nós configuramos os decorators para simplificar um ViewHelper e um Label:

```php
<?php
$form->setElementDecorators(array(
```

```
    'ViewHelper',
    'Label'
));
?>
```

Alguns Decorators são Inapropriados para Alguns Elementos

Enquanto `setElementDecorators()` pode parecer uma boa solução, há alguns casos que podem normalmente terminar com resultados inesperados. Por exemplo, os vários elementos button (Submit, Button, Reset) atualmente usam o rótulo como o valor do button, e somente usam os decorators ViewHelper e DtDdWrapper – prevenindo um rótulo adicional, erros e dica de serem renderizados; o exemplo acima duplicaria algum conteúdo (o rótulo).

Assim, use esse método sabiamente, e tenha consciência de que você pode precisar alterar manualmente alguns decorators de elementos para prevenir saída não desejada.

Exemplo 15.4. Configurar Filtros para Todos os Elementos

Em muitos casos, você pode querer aplicar o mesmo filtro para todos os elementos; um caso comum é para todos os valores de `trim()`:

```
<?php
$form->setElementFilters(array('StringTrim'));
?>
```

15.4.2.3. Métodos Para Interagir Com Elementos

Os seguintes métodos podem ser usados para interagir com elementos:

- `createElement($element, $name = null, $options = null)`
- `addElement($element, $name = null, $options = null)`
- `addElements(array $elements)`
- `setElements(array $elements)`
- `getElement($name)`
- `getElements()`
- `removeElement($name)`
- `clearElements()`
- `setDefaults(array $defaults)`
- `setDefault($name, $value)`
- `getValue($name)`
- `getValues()`
- `getUnfilteredValue($name)`

- `getUnfilteredValues()`
- `setElementFilters(array $filters)`
- `setElementDecorators(array $decorators)`
- `addElementPrefixPath($prefix, $path, $type = null)`
- `addElementPrefixPaths(array $spec)`

15.4.3. Grupos de Exibição

Grupos de exibição são um modo de criar agrupamentos virtuais de elementos para propósitos de exibição. Todos os elementos permanecem acessíveis pelo nome no formulário, mas quando iteramos sobre o formulário ou o renderizamos, quaisquer elementos em um grupo de exibição são renderizados juntos. O caso de uso mais comum para isso é para agrupar elementos em conjuntos de campos.

A classe base para grupos de exibição é `Zend_Form_DisplayGroup`. Enquanto ela pode ser instanciada diretamente, é tipicamente melhor usar o método `addDisplayGroup()` de `Zend_Form` para fazer isso. Esse método toma uma matriz de elementos como seu primeiro argumento, e um nome para o grupo de exibição como seu segundo argumento. Você pode opcionalmente passar uma matriz de opções ou um objeto `Zend_Config` como o terceiro argumento.

Assumindo que os elementos 'username' e 'password' estão configurados no formulário, o seguinte código agruparia esses elementos em um grupo de exibição 'login':

```
<?php
$form->addDisplayGroup(array('username', 'password'), 'login');
?>
```

Você pode acessar grupos de exibição usando o método `getDisplayGroup()`, ou via sobrecarga usando o nome do grupo de exibição:

```
<?php
// Using getDisplayGroup():
$login = $form->getDisplayGroup('login');
// Using overloading:
$login = $form->login;
?>
```

Decorators Padrão Não Precisam Ser Carregados

Por padrão, os decorators padrão são carregados durante a inicialização do objeto. Você pode desabilitar isso passando a opção 'disableLoadDefaultDecorators' quando criar um grupo de exibição:

```
<?php
$form->addDisplayGroup(
    array('foo', 'bar'),
    'foobar',
    array('disableLoadDefaultDecorators' => true)
```

```
);
```

Essa opção pode ser misturada com outras opções que você passar, tanto como opções de matriz ou em um objeto `Zend_Config`.

15.4.3.1. Operações Globais

Assim como acontece com os elementos, há algumas operações que podem afetar todos os grupos de exibição; essas incluem configurar decorators e configurar o caminho de plugin no qual procurar por decorators.

Exemplo 15.5. Configurar Caminho de Prefixo de Decorator para Todos os Grupos de Exibição

Por padrão, grupos de exibição herdam de qualquer caminhos de decorator que o formulário usar; entretanto, se eles devem procurar em locais alternativos, você pode usar o método `addDisplayGroupPrefixPath()`.

```
<?php
$form->addDisplayGroupPrefixPath('My_Foo_Decorator', 'My/Foo/Decorator');
?>
```

Exemplo 15.6. Configurar Decorators para Todos os Grupos de Exibição

Você pode configurar os decorators para todos os grupos de exibição. `setDisplayGroupDecorators()` aceita uma matriz de decorators, assim como `setDecorators()`, e sobrecreverá quaisquer decorators previamente configurados em cada grupo de exibição. Neste exemplo, nós configuramos os decorators para simplificar um conjunto de campos (o decorator FormElements é necessário para garantir que os elementos são iterados):

```
<?php
$form->setDisplayGroupDecorators(array(
    'FormElements',
    'Fieldset'
));
?>
```

15.4.3.2. Usando Classes de Grupo de Exibição Customizadas

Por padrão, `Zend_Form` usa a classe `Zend_Form_DisplayGroup` para exibir grupos. Você pode achar necessário estender essa classe de modo a fornecer funcionalidade customizada. `addDisplayGroup()` não permite a passagem de uma instância concreta, mas permite especificar a classe para usar como uma de suas opções, usando a chave 'displayGroupClass':

```
<?php
// Use a classe 'My_DisplayGroup'
$form->addDisplayGroup(
```

```
    array('username', 'password'),
    'user',
    array('displayGroupClass' => 'My_DisplayGroup')
);
?>
```

Se a classe não tem sido carregada, `Zend_Form` tentará fazer isso usando `Zend_Loader`.

Você pode também especificar uma classe de grupo de exibição padrão para usar com o formulário tal que todos os grupos de exibição criados com o objeto formulário usarão essa classe:

```
<?php
// Use a classe 'My_DisplayGroup' para todos os grupos de exibição:
$form->setDefaultDisplayGroupClass('My_DisplayGroup');
?>
```

Essa configuração pode ser especificada em configurações como 'defaultDisplayGroupClass', e será carregada mais cedo para garantir que todos os grupos de exibição usem essa classe.

15.4.3.3. Métodos para Interagir Com Grupos de Exibição

Os seguintes métodos podem ser usados para interagir com grupos de exibição:

- `addDisplayGroup(array $elements, $name, $options = null)`
- `addDisplayGroups(array $groups)`
- `setDisplayGroups(array $groups)`
- `getDisplayGroup($name)`
- `getDisplayGroups()`
- `removeDisplayGroup($name)`
- `clearDisplayGroups()`
- `setDisplayGroupDecorators(array $decorators)`
- `addDisplayGroupPrefixPath($prefix, $path)`
- `setDefaultDisplayGroupClass($class)`
- `getDefaultDisplayGroupClass($class)`

15.4.3.4. Métodos Zend_Form_DisplayGroup

`Zend_Form_DisplayGroup` tem os seguintes métodos, agrupados por tipo:

- Configuração:
    - `setOptions(array $options)`
    - `setConfig(Zend_Config $config)`
- Metadados:
    - `setAttrib($key, $value)`
    - `addAttribs(array $attribs)`

- setAttribs(array $attribs)
- getAttrib($key)
- getAttribs()
- removeAttrib($key)
- clearAttribs()
- setName($name)
- getName()
- setDescription($value)
- getDescription()
- setLegend($legend)
- getLegend()
- setOrder($order)
- getOrder()

- Elementos:
  - createElement($type, $name, array $options = array())
  - addElement($typeOrElement, $name, array $options = array())
  - addElements(array $elements)
  - setElements(array $elements)
  - getElement($name)
  - getElements()
  - removeElement($name)
  - clearElements()
- Plugin loaders:
  - setPluginLoader(Zend_Loader_PluginLoader $loader)
  - getPluginLoader()
  - addPrefixPath($prefix, $path)
  - addPrefixPaths(array $spec)
- Decorators:
  - addDecorator($decorator, $options = null)
  - addDecorators(array $decorators)
  - setDecorators(array $decorators)
  - getDecorator($name)
  - getDecorators()
  - removeDecorator($name)
  - clearDecorators()

- Renderização:
    - `setView(Zend_View_Interface $view = null)`
    - `getView()`
    - `render(Zend_View_Interface $view = null)`
- I18N:
    - `setTranslator(Zend_Translate_Adapter $translator = null)`
    - `getTranslator()`
    - `setDisableTranslator($flag)`
    - `translatorIsDisabled()`

15.4.4. Subformulários

Subformulários servem para diversos propósitos:

- Criar grupos de elementos lógicos. Uma vez que subformulários são simplesmente formulários, você pode validar subformulários como entidades individuais.
- Criar formulários multipágina. Uma vez que subformulários são simplesmente formulários, você pode exibir um subformulário separado por página, construindo formulários muitipágina onde cada formulário tem sua própria lógica de validação. Somente uma vez a validação de todos os subformulários fariam o formulário ser considerado completo.
- Agrupamentos de exibição. Como grupos de exibição, subformulários, quando renderizados como parte de um formulário maior, podem ser usados para agrupar elementos. Esteja atento, entretanto, que o objeto formulário mestre não terá consciência dos elementos em subformulários.

Um subformulário pode ser um objeto `Zend_Form`, ou, mais tipicamente, um objeto `Zend_Form_SubForm`. O último contém decorators que servem para inclusão em um formulário maior (por exemplo, ele não renderiza tags de formulário HTML adicionais, mas renderiza grupos de elementos). Para anexar um subformulário, simplesmente adicione-o ao formulário e dê-lhe um nome:

```
<?php
$form->addSubForm($subForm, 'subform');
?>
```

Você pode recuperar um subformulário usando ou `getSubForm($name)` ou sobrecarregando o nome do subformulário:

```
<?php
// Usando getSubForm():
$subForm = $form->getSubForm('subform');
// Usando sobrecarga:
$subForm = $form->subform;
?>
```

Subformulários são incluídos na iteração do form, mesmo que os elementos que ele contém não

sejam.

15.4.4.1. Operações Globais

Como elementos e grupos de exibição, há algumas operações que podem precisar afetar todos os subformulários. Ao contrário de grupos de exibição e elementos, entretanto, subformulários herdam a maioria das funcionalidades do objeto formulário mestre, e somente a operação real que possa precisar ser executada globalmente está configurando decorator para subformulários. Para esse propósito, há o método `setSubFormDecorators()`. No próximo exemplo, nós configuraremos o decorator para todos os subformulários a serem simplesmente um conjunto de campos (o decorator FormElements é necessário para garantir que seus elementos sejam iterados):

```
<?php
$form->setSubFormsDecorators(array(
    'FormElements',
    'Fieldset'
));
?>
```

15.4.4.2. Métodos para Interagir Com Subformulários

Os seguintes métodos podem ser usados para interagir com subformulários:

- `addSubForm(Zend_Form $form, $name, $order = null)`
- `addSubForms(array $subForms)`
- `setSubForms(array $subForms)`
- `getSubForm($name)`
- `getSubForms()`
- `removeSubForm($name)`
- `clearSubForms()`
- `setSubFormDecorators(array $decorators)`

15.4.5. Metadados e Atributos

Enquanto uma utilidade do formulário primariamente deriva dos elementos que ele contém, ele pode conter outros metadados, tais como um nome (freqüentemente usado como um ID único na marcação HTML); os atributos action e method; o número de elementos, grupos, e subformulários que ele contém; e arbitrariamente metadados (geralmente usados para configurar atributos HTML para a própria tag form).

Você pode configurar e recuperar um nome de formulário usando os acessores de nome:

```
<?php
// Configura o nome:
$form->setName('registration');
// Recupera o nome:
$name = $form->getName();
?>
```

Para configurar o atributo action (url para a qual o formulário submete) e o atributo method (método pelo qual deverá submeter, por exemplo, 'POST' ou 'GET'), use os acessores de action e method:

```
<?php
// Configura os atributos action e method:
$form->setAction('/user/login')
     ->setMethod('post');
?>
```

Nota

Os atributos method e action são usados somente internamente para renderizar, e não para qualquer tipo de validação.

`Zend_Form` implementa a interface `Countable`, permitindo a você passá-lo como um argumento para contar:

```
<?php
$numItems = count($form);
?>
```

A configuração abritrária de metadados é feita através de acessores de atributos. Uma vez que a sobrecarga em `Zend_Form` é usada para acessar elementos, grupos de exibição, e subformulários, esse é o único método para acessar metadados.

```
<?php
// Configura atributos:
$form->setAttrib('class', 'zend-form')
     ->addAttribs(array(
         'id'       => 'registration',
         'onSubmit' => 'validate(this)',
     ));
// Recupera atributos:
$class = $form->getAttrib('class');
$attribs = $form->getAttribs();
// Remove um atributo:
$form->removeAttrib('onSubmit');
// Limpa todos os atributos:
$form->clearAttribs();
?>
```

15.4.6. Decorators

Criar uma marcação para um formulário é freqüentemente uma tarefa consumidora de tempo, particularmente se você planeja reutilizar a mesma marcação para mostrar coisas tais como erros de validação, valores submetidos, etc. A resposta de `Zend_Form` para esse problema é *decorators*.

Decorators para objetos `Zend_Form` podem ser usados para renderizar um formulário. O decorator FormElements irá iterar através de todos os itens em um formulário – elementos, grupos de exibição, e subformulários – e renderizá-los, retornando o resultado. Decorators adicionais podem

ser então usados para envolver esse conteúdo, ou adicioná-lo ou incluir algo em seu início.

Os decorator padrão para `Zend_Form` são FormElements, HtmlTag (encapsulado em uma lista de definição), e Form; o código equivalente para criá-los é o seguinte:

```
<?php
$form->setDecorators(array(
    'FormElements',
    array('HtmlTag', array('tag' => 'dl')),
    'Form'
));
?>
```

Isso cria saída como a seguinte:

```
<form action="/form/action" method="post">
<dl>
...
</dl>
</form>
```

Qualquer conjunto de atributos no formulário será usado como atributos HTML da tag `<form>`.

Decorators Padrão Não Precisam Ser Carregados

Por padrão, os decorators padrão são carregados durante a inicialização do objeto. Você pode desabilitar isso passando a opção 'disableLoadDefaultDecorators' para o construtor:

```
<?php
$form = new Zend_Form(array('disableLoadDefaultDecorators' => true));
```

Essa opção pode ser misturada com quaisquer outras opções que você passar, tanto como opções de matriz ou em um objeto `Zend_Config`.

Usando Decorators Múltiplos do Mesmo Tipo

Internamente, `Zend_Form` usa uma classe decorator como mecanismo de busca quando recupera decoratros. Como um resultado, você não pode registrar múltiplos decorators do mesmo tipo; decorators subseqüentes simplesmente sobrescreverão aqueles que existiam antes.

Para contornar isso, você pode usar apelidos. Ao invés de passar um decorator ou o nome do decorator como o primeiro argumento para `addDecorator()`, passe uma matriz com um elemento simples, com o apelido apontando para o o objeto ou nome do decorator:

```
<?php
// Apelido para 'FooBar':
$form->addDecorator(array('FooBar' => 'HtmlTag'), array('tag' => 'div'));
// E recupera mais tarde:
$form = $element->getDecorator('FooBar');
?>
```

Nos métodos `addDecorators()` e `setDecorators()`, você irá precisar passar a opção 'decorator' na matriz representando o decorator:

```
<?php
// Adiciona dois decorators 'HtmlTag'  apelidando um como 'FooBar':
$form->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);
// E recupera mais tarde:
$htmlTag = $form->getDecorator('HtmlTag');
$fooBar  = $form->getDecorator('FooBar');
?>
```

Você pode criar seus próprios decorators para gerar o formulário. Um caso de uso comum é se você conhece o HTML exato que deseja usar; seu decorator poderia criar o HTML exato e simplesmente retorná-lo, potencialmente usando os decorators dos elementos individuais ou grupos de exibição.

Os seguintes métodos podem ser usados para interagir com decorators:

- `addDecorator($decorator, $options = null)`
- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)`
- `getDecorator($name)`
- `getDecorators()`
- `removeDecorator($name)`
- `clearDecorators()`

15.4.7. Validação

Um caso de uso primários para formulários é validar dados submetidos. `Zend_Form` permite que você valide um formulário inteiro de uma vez ou um formulário parcial, assim como automatize respostas de validação para XmlHttpRequests (AJAX). Se o dado submetido não é válido, ele tem métodos para recuperar os vários códigos e mensagens de erro para falhas de validação de elementos e subformulários.

Para validar um formulário inteiro, use o método `isValid()`:

```
<?php
if (!$form->isValid($_POST)) {
    // falha na validação
}
?>
```

`isValid()` irá validar cada elemento obrigatório, e qualquer elemento não obrigatório contido nos dados submetidos.

Algumas vezes você precisa validar somente um subconjunto dos dados; para isso, use `isValidPartial($data)`:

```
<?php
if (!$form->isValidPartial($data)) {
    // falha na validação
}
?>
```

`isValidPartial()` tenta somente validar aqueles itens nos dados para os quais há elementos que casam; se m elemento não está representado nos dados, ele é pulado.

Quando validar elementos ou grupos de elementos para uma requisição AJAX, você tipicamente estará validando um subconjunto do formulário, e quer a resposta de volta em JSON. `processAjax()` faz precisamente isso:

```
<?php
$json = $form->processAjax($data);
?>
```

Você pode então simplesmente enviar uma resposta JSON para o cliente. Se o formulário é válido, haverá um resposta booleana true. Se não, será um objeto javascript contendo pares chave/mensagem, onde cada 'mensagem' é uma matriz de mensagens de erro de validação.

Para formulários que falham na validação, você pode recuperar tanto códigos quanto mensagens de erro, usando `getErrors()` e `getMessages()`, respectivamente:

```
<?php
$codes = $form->getErrors();
$messages = $form->getMessage();
?>
```

Nota

Uma vez que as mensagens retornadas por `getMessages()` são uma matriz de pares código/mensagem, `getErrors()` tipicamente não é necessária.

Você pode recuperar códigos e mensagens de erro para elementos individuais pela simples passagem do nome do elemento:

```
<?php
$codes = $form->getErrors('username');
$messages = $form->getMessages('username');
?>
```

Nota

Nota: Quando validar elementos, `Zend_Form` envia um segundo argumento para cada

método `isValid()` de elemento: a matriz de dados a ser validada. Isso pode então ser usado por validadores individuais para permitir a eles utilizar outros valores submetidos quando determinar a validade dos dados. Um exemplo seria um formulário de registro que requer tanto uma senha quanto uma confirmação de senha; o elemento senha poderia usar o elemento confirmação como parte de sua validação.

15.4.8. Métodos

A seguir uma lista comleta de métodos disponíveis para `Zend_Form`, agrupados por tipo:

- Configuração e Opções:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- Plugin Loaders e caminhos:
  - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type = null)`
  - `getPluginLoader($type = null)`
  - `addPrefixPath($prefix, $path, $type = null)`
  - `addPrefixPaths(array $spec)`
  - `addElementPrefixPath($prefix, $path, $type = null)`
  - `addElementPrefixPaths(array $spec)`
  - `addDisplayGroupPrefixPath($prefix, $path)`
- Metadados:
  - `setAttrib($key, $value)`
  - `addAttribs(array $attribs)`
  - `setAttribs(array $attribs)`
  - `getAttrib($key)`
  - `getAttribs()`
  - `removeAttrib($key)`
  - `clearAttribs()`
  - `setAction($action)`
  - `getAction()`
  - `setMethod($method)`
  - `getMethod()`
  - `setName($name)`
  - `getName()`
- Elementos:
  - `addElement($element, $name = null, $options = null)`
  - `addElements(array $elements)`

- setElements(array $elements)
- getElement($name)
- getElements()
- removeElement($name)
- clearElements()
- setDefaults(array $defaults)
- setDefault($name, $value)
- getValue($name)
- getValues()
- getUnfilteredValue($name)
- getUnfilteredValues()
- setElementFilters(array $filters)
- setElementDecorators(array $decorators)

- Subformulários:
  - addSubForm(Zend_Form $form, $name, $order = null)
  - addSubForms(array $subForms)
  - setSubForms(array $subForms)
  - getSubForm($name)
  - getSubForms()
  - removeSubForm($name)
  - clearSubForms()
  - setSubFormDecorators(array $decorators)
- Grupos de Exibição:
  - addDisplayGroup(array $elements, $name, $options = null)
  - addDisplayGroups(array $groups)
  - setDisplayGroups(array $groups)
  - getDisplayGroup($name)
  - getDisplayGroups()
  - removeDisplayGroup($name)
  - clearDisplayGroups()
  - setDisplayGroupDecorators(array $decorators)
- Validação
  - populate(array $values)
  - isValid(array $data)
  - isValidPartial(array $data)
  - processAjax(array $data)

    - `persistData()`
    - `getErrors($name = null)`
    - `getMessages($name = null)`
- Renderização:
    - `setView(Zend_View_Interface $view = null)`
    - `getView()`
    - `addDecorator($decorator, $options = null)`
    - `addDecorators(array $decorators)`
    - `setDecorators(array $decorators)`
    - `getDecorator($name)`
    - `getDecorators()`
    - `removeDecorator($name)`
    - `clearDecorators()`
    - `render(Zend_View_Interface $view = null)`
- I18N:
    - `setTranslator(Zend_Translate_Adapter $translator = null)`
    - `getTranslator()`
    - `setDisableTranslator($flag)`
    - `translatorIsDisabled()`

15.4.9. Configuração

`Zend_Form` é completamente configurável via `setOptions()` e `setConfig()` (ou pela passagem de opções ou de um objeto `Zend_Config` para o construtor). Usando esses métodos, você pode especificar elementos de formulário, grupos de exibição, decorators e metadados.

Como uma regra geral, se 'set' + a chave de opção refere-se a um método `Zend_Form`, então o valor fornecido será passado para aquele método.

Exceções à regra incluem o seguinte:

- `prefixPaths` será passado para `addPrefixPaths()`
- `elementPrefixPaths` será passado para `addElementPrefixPaths()`
- `displayGroupPrefixPaths` será passado para `addDisplayGroupPrefixPaths()`
- os seguintes modificadores não podem ser configurados desse modo:
    - `setAttrib (though setAttribs *will* work)`
    - `setConfig`
    - `setDefault`
    - `setOptions`
    - `setPluginLoader`

- setSubForms
- setTranslator
- setView

Como um exemplo, aqui está um arquivo de configuração que passa a configuração para cada tipo de dado configurável:

```
[element]
name = "registration"
action = "/user/register"
method = "post"
attribs.class = "zend_form"
attribs.onclick = "validate(this)"
disableTranslator = 0
prefixPaths.element.prefix = "My_Element"
prefixPaths.element.path = "My/Element/"
elementPrefixPaths.validate.prefix = "My_Validate"
elementPrefixPaths.validate.path = "My/Validate/"
displayGroupPrefixPaths.prefix = "My_Group"
displayGroupPrefixPaths.path = "My/Group/"
elements.username.type = "text"
elements.username.options.label = "Username"
elements.username.options.validators.alpha.validator = "Alpha"
elements.username.options.filters.lcase = "StringToLower"
; more elements, of course...
elementFilters.trim = "StringTrim"
;elementDecorators.trim = "StringTrim"
displayGroups.login.elements.username = "username"
displayGroups.login.elements.password = "password"
displayGroupDecorators.elements.decorator = "FormElements"
displayGroupDecorators.fieldset.decorator = "Fieldset"
decorators.elements.decorator = "FormElements"
decorators.fieldset.decorator = "FieldSet"
decorators.fieldset.decorator.options.class = "zend_form"
decorators.form.decorator = "Form"
?>
```

O código acima poderia facilmente ser abstraído para um arqivo de configuração XML ou PHP baseado em matriz.

15.4.10. Formulário Customizados

Uma alternativa para usar formulários baseados em configuração é estender `Zend_Form`. Isso tem vários benefícios:

- Você pode testar unitariamente seu formulário facilmente para garantir validações e renderizar a execução como esperado.
- Controle granulado elegantemente sobre elementos individuais.
- Reuso de objetos de formulário, e alta portabilitade (sem necessidade de rastrear arquivos de configuração).
- Implementação de funcionalidade customizada.

O caso de uso mais típico será usar o método `init()` para configurar elementos de formulário e configurações específicos:

```php
<?php
class My_Form_Login extends Zend_Form
{
    public function init()
    {
        $username = new Zend_Form_Element_Text('username');
        $username->class = 'formtext';
        $username->setLabel('Username:')
                 ->setDecorators(array(
                     array('ViewHelper', array('helper' => 'formText')),
                     array('Label', array('class' => 'label'))
                 ));
        $password = new Zend_Form_Element_Password('password');
        $password->class = 'formtext';
        $password->setLabel('Username:')
                 ->setDecorators(array(
                     array('ViewHelper', array('helper' => 'formPassword')),
                     array('Label', array('class' => 'label'))
                 ));
        $submit = new Zend_Form_Element_Submit('login');
        $submit->class = 'formsubmit';
        $submit->setValue('Login')
               ->setDecorators(array(
                   array('ViewHelper', array('helper' => 'formSubmit'))
               ));
        $this->addElements(array(
            $username,
            $password,
            $submit
        ));
        $this->setDecorators(array(
            'FormElements',
            'Fieldset',
            'Form'
        ));
    }
}
?>
```

Esse formulário pode então ser instanciado com simplesmente:

```php
<?php
$form = new My_Form_Login();
?>
```

e todos as funcionalidades já estarão configuradas e prontas; sem necessidade de arquivos de configuração. (Note que esse exemplo e extremamente simplificado, já que não contém validadores ou filtros para os elementos.)

Outra razão comum para extensão é definir um conjunto de decorators padrão. Você pode fazer isso pela sobrescrita do método `loadDefaultDecorators()`:

```php
<?php
class My_Form_Login extends Zend_Form
{
    public function loadDefaultDecorators()
```

```
    {
        $this->setDecorators(array(
            'FormElements',
            'Fieldset',
            'Form'
        ));
    }
}
```

15.5. Criando Marcações de Formulários Customizados usando Zend_Form_Decorator

Renderizar um formulário é completamente opcional – você não precisa usar os métodos render() de `Zend_Form` para tudo. Entretanto, se você usar, decorators serão usados para renderizar os diversos objetos de formulário.

Um número arbitrário de decorators podem ser anexados para cada item (elementos, grupos de exibição, subformulários, ou o próprio objeto de formulário); entretanto, somente um dado tipo pode ser anexado a cada item. Decorators são chamados na ordem que eles são registrados. Dependendo de um decorator, ele pode substituir o conteúdo passado a ele, ou adicionar conteúdo ou incluir conteúdo no início do mesmo.

O estado do objeto é configurado via opções de configuração passadas ao construtor ou método `setOptions()`. Quando criar decorators via um item de `addDecorator()` ou métodos relacionados, opções podem ser passadas como um argumento para o método. Essas podem ser usadas para especificar localização, um separador a ser usado entre o que foi passado no conteúdo e o conteúdo gerado recentemente, e quaisquer opções que o decorator suporte.

Antes de cada método `render()` do decorator ser chamado, o item atual está configurado no decorator usando `setElement()`, dando ao decorator a ciência do item sendo renderizado. Isso permite a você criar decorators que podem renderizar porções específicas do item – tais como rótulo, o valor, mensagens de erro, etc. Através do encadeamento de vários decorators que renderizam segmentos de elemento específicos, você pode construir marcação complexa representando o item inteiro.

15.5.1. Operação

Para configurar um decorator, passe uma matriz de opções ou um objeto `Zend_Config` para su construtor, uma matriz para `setOptions()`, ou um objeto `Zend_Config` para `setConfig()`.

Opções padrão incluem:

- `placement`: Placement (colocação) pode ser 'append' ou 'prepend' (case insensitive), e indica se o conteúdo passado para `render()` será adicionado ou incluído no início, respectivamente. No caso em que um decorator substitui o conteúdo, essa configuração é ignorada. A configuração padrão é append.
- `separator`: O separador é usado entre o conteúdo passado para `render()` e o novo conteúdo gerado pelo decorator, ou entre os itens renderizados pelo decorator (por exemplo, FormElements usam o separador entre cada item renderizado). No caso em que um decorator substitui o conteúdo, essa configuração pode ser ignorada. O valor padrão é

PHP_EOL.

A interface do decorator especifica métodos para interagir com opções. Esses incluem:

- `setOption($key, $value)`: configura uma opção simples.
- `getOption($key)`: recupera um valor de opção simples.
- `getOptions()`: recupera todas as opções.
- `removeOption($key)`: remove uma opção simples.
- `clearOptions()`: remove todas as opções.

Decorators são significativos para interagir com os vários tipos de classe `Zend_Form`: `Zend_Form`, `Zend_Form_Element`, `Zend_Form_DisplayGroup`, e todas as classes derivadas delas. O método `setElement()` permite a você configurar o objeto com o qual o decorator está trabalhando, e `getElement()` é usado para recuperá-lo.

Cada método `render()` de decorator aceita uma string, `$content`. Quando o primeiro decorator é chamado, essa string está tipicamente vazia, enquanto em chamadas subseqüentes será populada. Baseado no tipo de decorator e nas opções passadas, o decorator substituirá essa string, incluirá no início da string, ou adicionará à string; um separador opcional será usado nas duas últimas situações.

15.5.2. Decorators Padrão

`Zend_Form` vem embracado com muitos decorators padrão; veja o capítulo sobre Decorators Padrão para detalhes.

15.5.3. Decorators Customizados

Se você achar que sua renderização precisa ser complexa ou precisa de customização pesada, você deve considerar a criação de um decorator customizado.

Decorators precisams somente implementar `Zend_Decorator_Interface`. A interface especifica o seguinte:

```
<?php
interface Zend_Decorator_Interface
{
    public function __construct($options = null);
    public function setElement($element);
    public function getElement();
    public function setOptions(array $options);
    public function setConfig(Zend_Config $config);
    public function setOption($key, $value);
    public function getOption($key);
    public function getOptions();
    public function removeOption($key);
    public function clearOptions();
    public function render($content);
}
?>
```

Para fazer isso mais imples, você pode simplesmente estender `Zend_Decorator_Abstract`,

que implementa todos os métodos exceto `render()`.

Como um exemplo, digamos que você quer reduzir o número de decorators que você usa, e construir um decorator "composite" que tomará conta de renderizar o rótulo, elemento, quaisquer mensagens de erro, e descrição em uma `div` HTML. Você poderia construir tal decorator 'Composite' como segue:

```
<?php
class My_Decorator_Composite extends Zend_Form_Decorator_Abstract
{
    public function buildLabel()
    {
        $element = $this->getElement();
        $label = $element->getLabel();
        if ($translator = $element->getTranslator()) {
            $label = $translator->translate($label);
        }
        if ($element->getRequired()) {
            $label .= '*';
        }
        $label .= ':';
        return $element->getView()->formLabel($element->getName(), $label);
    }
    public function buildInput()
    {
        $element = $this->getElement();
        $helper  = $element->helper;
        return $element->getView()->$helper(
            $element->getName(),
            $element->getValue(),
            $element->getAttribs(),
            $element->options
        );
    }
    public function buildErrors()
    {
        $element  = $this->getElement();
        $messages = $element->getMessages();
        if (empty($messages)) {
            return '';
        }
        return '<div class="errors">' . $element->getView()-
>formErrors($messages) . '</div>';
    }
    public function buildDescription()
    {
        $element = $this->getElement();
        $desc    = $element->getDescription();
        if (empty($messages)) {
            return '';
        }
        return '<div class="description">' . $desc . '</div>';
    }
    public function render($content)
    {
        $element = $this->getElement();
        if (!$element instanceof Zend_Form_Element) {
            return $content;
        }
        if (null === $element->getView()) {
```

```
            return $content;
        }
        $separator = $this->getSeparator();
        $placement = $this->getPlacement();
        $label     = $this->buildLabel();
        $input     = $this->buildInput();
        $errors    = $this->buildErrors();
        $desc      = $this->buildDescription();
        $output = '<div class="form element">'
                . $label
                . $input
                . $errors
                . $desc
                . '</div>'
        switch ($placement) {
            case (self::PREPEND):
                return $output . $separator . $content;
            case (self::APPEND):
            default:
                return $content . $separator . $output;
        }
    }
}
?>
```

Você pode então colocar isso no caminho do decorator:

```
<?php
// para um elemento:
$element->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
// para todos os elementos:
$form->addElementPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
?>
```

Você pode então especificar esse decorator como 'Composite' e anexá-lo ao elemento:

```
<?php
// sobrescreve decorators existentes com esse:
$element->setDecorators(array('Composite'));
?>
```

Enquanto esse exemplo mostrou como criar um decorator que renderiza saída complexa de várias propriedades de elemento, você pode também criar decorators que manipulam um aspecto simples de um elemento; os decorators 'Decorator' e 'Label' são excelentes exemplos dessa prática. Fazer isso permite a você misturar e encaixar decorators para obter saída complexa – e também sobrescrever aspectos simples de decoração a customizar para suas necessidades.

Por exemplo, se você quer simplesmente exibir uma mensagem que um erro ocorreu quando validou um elemento, mas não exibir cada uma das mensagens de erro de validação individuais, você poderia criar seu próprio 'Errors' decorator:

```
<?php
class My_Decorator_Errors
{
```

```
    public function render($content = '')
    {
        $output = '<div class="errors">The value you provided was invalid;
            please try again</div>';
        $placement = $this->getPlacement();
        $separator = $this->getSeparator();
        switch ($placement) {
            case 'PREPEND':
                return $output . $separator . $content;
            case 'APPEND':
            default:
                return $content . $separator . $output;
        }
    }
}
?>
```

Nesse elemento particular, por causa do segmento final do decorator, 'Errors', casar o mesmo que `Zend_Form_Decorator_Errors`, será renderizado *no lugar* daquele decorator – significando que você não precisaria alterar quaisquer decorators para modificar a saída. Através da nomeação de seus decorators depois de decorators padrão existentes, você pode modificar decoração sem precisar modificar seus decorators de elementos.

15.6. Elementos de Formulário Padrão Embarcados Com Zend Framework

Zend Framework vem embarcado com classes de elemento concretas cobrindo a maioria dos elementos de formulário HTML. A maioroa simplesmente especifica um view helper particular para usar quando decorar o elemento, mas vários oferecem funcionalidades adicionais. O que se segue é uma lista de todas essas classes, assim como as descrições de funcionalidade que elas oferecem.

15.6.1. Zend_Form_Element_Button

Usada para criar elementos de botão HTML, `Zend_Form_Element_Button` estende Zend_Form_Element_Submit, derivando sua funcionalidade customizada. Ela especifica o view helper 'formButton' para decoração.

Como o elemento submit, ela usa o rótulo do element ocomo valor do elemento para propósitos de exibição; em outras palavras, para configurar o texto do botão, configure o valor do elemento. O rótulo será traduzido se o adaptador de tradução estiver presente.

Porque o rótulo é usado como parte do elemento, o elemento botão usa somente os decorators ViewHelper e DtDdWrapper.

Depois de popular ou validar um formulário, você pode verificar se o botão dado foi clicado usando o método`isChecked()`.

15.6.2. Zend_Form_Element_Checkbox

Caixas de verificação HTML permitem a você retornar retornar um valor específico, mas basicamente operar como booleanos: quando está verificado, o valor é submetido; quando não está verificado, nada é submetido. Internamente, Zend_Form_Element_Checkbox força esse estado.

Por padrão, o valor verificado é '1', e o valor não verificado é '0'. Você pode especificar os valores a

serem usados com os acessores `setCheckedValue()` e `setUncheckedValue()` respectivamente. Internamente, a qualquer momento você configura o valor, se o valor fornecido casar com o valor verificado, então ele é configurado, mas quaisquer outros valores provocam a não verificação do valor a ser configurado.

Adicionalmente, configurar o valor configura a propriedade `checked` da caixa de verificação. Você pode consultar isso usando `isChecked()` ou simplesmente acessar a propriedade. Usar o método `setChecked($flag)` irá tanto configurar o estado do marco como configurar o valor apropriado (verificado ou não verificado) no elemento. Por favor, use esse método quando configurar o estado verificado de um elemento caixa de verificação para garantir que o valor foi configurado apropriadamente.

`Zend_Form_Element_Checkbox` usa o view helper 'formCheckbox'. O valor verificado é sempre usado para populá-lo.

15.6.3. Zend_Form_Element_Hidden

Elementos ocultos somente injetam dados que devem ser submetidos, mas que o usuário não deve manipular. `Zend_Form_Element_Hidden` efetua isso através do uso do view helper 'formHidden'.

15.6.4. Zend_Form_Element_Hash

Esse elemento fornece proteção de ataques CSRF (Cross Site Request Forgeries) sobre formulários, garantindo que os dados foram submetidos pela sessão do usuário que gerou o formulário e não por um script vampiro. Proteção é obtida adicionando um elemento hash ao formulário e verificando-o quando o formulário é submetido.

O nome do elemento hash deve ser único. É recomendado usar a opção `salt` para o elemento, dois hashes com o mesmo nome e diferentes salts não colidirem:

```
<?php
$form->addElement('hash', 'no_csrf_foo', array('salt' => 'unique'));
?>
```

Você pode configurar o último salt usando o método `setSalt($salt)`.

Internamente, o elemento armazena um identificador único usando `Zend_Session_Namespace`, e verifica-o na submissão (verificando que o TTL não expirou). O validador 'Identical' é então usado para garantir que o hash submetido casa com o hash aramazenado.

O view helper 'formHidden' é usado para renderizar o elemento no formulário.

15.6.5. Zend_Form_Element_Image

Imagens podem ser usadas como elementos de formulário, e permitem a você especificar elementos gráficos como botões de formulário.

Elementos imagem precisam de uma fonte de imagem. `Zend_Form_Element_Image` permite a você especirficar isso pelo uso do acessor `setImage()` (ou chave de configuração 'image'). Você pode também opcionalmente especificar um valor a ser usado quando submeter a imagem usando o

acessor `setImageValue()` (ou a chave de configuração 'imageValue'). Quando o valor configurado para o elemento casar com `imageValue`, então o acessor `isChecked()` retornará true.

O elemento imagem usa Image Decorator para renderizar (assim como os decorators padrão Errors, HtmlTag, e Label). Você pode opcionalmente especificar uma tag para o decorator `Image` que irá encapsular o elemento imagem.

15.6.6. Zend_Form_Element_MultiCheckbox

Freqüentemente você tem um conjunto de caixas de verificação relacionadas, e você deseja um grupo de resoltados. Isso é mais como um Multiselect, mas ao invés deles estarem em uma lista dropdown, você precisa mostrar pares caixa de verificação/valor.

`Zend_Form_Element_MultiCheckbox` faz isso em um estalo. Como todos os outros elementos estendem a base Multielemento, você pode especificar uma lista de opções, e facilmente validar contra a mesma lista. O view helper 'formMultiCheckbox' garante que essas são retornadas como uma matriz na submissão do formulário.

Você pode manipular as várias opções da caixa de verificação usando os seguintes métodos:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (sobrescreve as opções existentes)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

15.6.7. Zend_Form_Element_Multiselect

Elementos `select` XHTML permitem um atributo 'múltiplo', indicando que múltiplas opções podem ser selecionadas para submissão, aon invés de uma. `Zend_Form_Element_Multiselect` estende Zend_Form_Element_Select, e configura o atributo `multiple` para 'multiple'. Como outras classes que herdam da classe base `Zend_Form_Element_Multi`, você pode manipular as opções para o seleção usando:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (sobrescreve as opções existentes)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

Se o adaptador de tradução estiver registrado com o formulário e/ou elemento, valores de opção serão traduziados para propósitos de exibição.

15.6.8. Zend_Form_Element_Password

Elementos de senha são basicamente elementos de texto normais – exceto pelo fato de que você tipicamente não quer a senha submetida exibida em mensagens de erro ou no próprio elemento quando o formulário é exibido novamente.

`Zend_Form_Element_Password` obtém isso pela chamada a `setObscureValue(true)` em cada validador (garantindo que a senha seja obscurecida em mensagens de erro de validação), e usando o view helper 'formPassword' (que não mostrará o valor passado para ele).

15.6.9. Zend_Form_Element_Radio

Elementos Radio permitem a você especificar várias opções, das quais você precisa que um único valor seja retornado. `Zend_Form_Element_Radio` estende a classe base `Zend_Form_Element_Multi`, permitindo a você especificar um número de opções, e então usar o view helper `formRadio` para exibi-las.

Como todos os elementos estendendo a classe base Multielemento, os seguintes métodos podem ser usados para manipular as opções radio exibidas:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (sobrescreve as opções existentes)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

15.6.10. Zend_Form_Element_Reset

Botões Reset são tipicamente usados para limpar um formulário, e não são parte dos dados submetidos. Entretanto, como eles servem um propósito na exibição, eles são incluídos nos elementos padrão.

`Zend_Form_Element_Reset` estende Zend_Form_Element_Submit. Com tal, o rótulo é usado para a exibição do botão, e será traduzido se um adaptador de tradução estiver presente. Ele utiliza somente os decorators 'ViewHelper' e 'DtDdWrapper', já que bubca haverá mensagens de erro para tais elementos, nem um rótulo será necessário.

15.6.11. Zend_Form_Element_Select

Caixas de seleção são um modo comum de limitar escolhas específicas para um dado de formulário. `Zend_Form_Element_Select` permite a você gerar isso rapidamente e facilmente.

Como estende o Multielemento base, os seguintes métodos podem ser usados para manipular as opções de seleção:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (sobrescreve opções existentes)

- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

`Zend_Form_Element_Select` usa o view helper 'formSelect' para decoração.

15.6.12. Zend_Form_Element_Submit

Botões Submit são usados para submeter um formulário. Você pode usar múltiplos botões submit; você pode usar o botão usado para submeter o formulário para decidir que ação tomará com os dados submetidos. `Zend_Form_Element_Submit` faz esse processo de decisão fácil, pela adição do método `isChecked()`; como somente um elemento de botão será submetido pelo formulário, depois de popular ou validar o formulário, você pode chamar esse método para cada botão submit para determinar qual deles foi usado.

`Zend_Form_Element_Submit` usa o rótulo como o "value" do botão submit, traduzindo-o se um adaptador de tradução estiver presente. `isChecked()` verifica o valor submetido contra o rótulo de modo a determinar se o botão foi usado.

Os decorators ViewHelper e DtDdWrapper renderizam o elemento. Nenhum decorator de rótulo é usado, como o rótulo de botão é usado quando renderizar o elemento; também, tipicamente, você não associará erros com um elemento submit.

15.6.13. Zend_Form_Element_Text

De longe o mais predominante tipo de elemento de formulário é o elemento texto, permitindo entrada de texto limitada; é um elemento ideal para a maior dos dados de entrada. `Zend_Form_Element_Text` simplesmente usa o view helper 'formText' para exibir o elemento.

15.6.14. Zend_Form_Element_Textarea

Textareas são usados quando grandes quantidades de texto são esperadas, e não há limites na quantidade de texto submetido (a não ser que o tamanho máximo limite como ditado pelo seu servidor ou PHP). `Zend_Form_Element_Textarea` usa o view helper 'textArea' para exibir tais elementos, colocando o valor como o conteúdo do elemento.

15.7. Decorators de Formulário Padrão Embarcados Com Zend Framework

`Zend_Form` vem embracado com diversos decorators padrão. Para mais informações sobre o uso geral de decorators, veja a seção Decorators.

15.7.1. Zend_Form_Decorator_Callback

O decorator Callback pode executar um callback arbitrário para renderizar conteúdo. Callbacks devem ser especificados via opção 'callback' passado na configuração do decorator, e podem ser qualquer tipo callback PHP válido. Callbacks devem aceitar três argumentos, `$content` (o conteúdo original passado para o decorator), `$element` (o item sendo decorado), e uma matriz de `$options`. Como um exemplo de callback:

```php
<?php
class Util
{
    public static function label($content, $element, array $options)
    {
        return '<span class="label">' . $element->getLabel() . "</span>";
    }
}
?>
```

Esse callback seria especificado como `array('Util', 'label')`, e geraria alguma (má) marcação de HTML para o rótulo. O decorator Callback então substituiria, acrescentaria ou adicionaria no início do conteúdo original o valor de retorno do primeiro.

O decorator Callback permite a você especificar um valor null para a opção de colocação, que irá substituir o conteúdo original com o valor de retorno do callback; 'prepend' e 'append' ainda são válidas.

15.7.2. Zend_Form_Decorator_Description

O decorator Description pode ser usado para exibir uma descrição configurada em um item `Zend_Form`, `Zend_Form_Element`, ou `Zend_Form_DisplayGroup`; ela puxa a descrição usando o método `getDescription()` do objeto. Casos de uso comum são para fornecer dicas UI[3] para seus elementos.

Por padrão, se nenhum descrição estiver presente, nenhuma saída é gerada. Se a descrição estiver presente, então é envolvida em uma tag HTML `p` por padrão, mesmo que você possa especificar uma tag pela passagem da opção `tag` quando criar o decorator, ou chamando `setTag()`. Você pode adicionalmente especificar uma classe para a tag usando a opção `class` ou chamando `setClass()`; por padrão, a classe 'hint' é usada.

A descrição é escapada[4] usando os mecanismos de escaping do objeto view por padrão. Você pode desabilitar isso passando um valor `false` para a opção 'escape' do ou método `setEscape()`.

15.7.3. Zend_Form_Decorator_DtDdWrapper

Os decorators padrão utilizam listas de definição (`<dl>`) para renderizar elementos de formulário. Uma vez que itens de formulário podem aparecer em qualquer ordem, grupos de exibição e subformulários podem ser intercalados com outros itens de formulário. Para manter esses tipos de item particulares dentro da lista de definição, o DtDdWrapper cria um novo termo de definição vazio (`<dt>`) e envolve seu conteúdo em uma nova definição de dados (`<dd>`). A saída parece alguma coisa como essa:

```
<dt></dt>
<dd><fieldset id="subform">
    <legend>User Information</legend>
    ...
</fieldset></dd>
```

Esse decorator substitui o conteúdo fornecido por ele pelo que está envolvido dentro do elemento

3 User Interface (Interface Gráfica).
4 (De novo esse termo horrível) Quer dizer que os comandos embutidos no texto são ignorados.

<dd>.

15.7.4. Zend_Form_Decorator_Errors

Elementos errors obtêm seu próprio decorator com o decorator Errors. Esse decorator atua como substituto para o view helper FormErrors, que renderiza mensagens de erro em uma lista não ordenada (<ul>) como itens de lista. O elemento <ul> recebe uma classe de "errors".

O decorator Errors pode adicionar o conteúdo fornecido para ele no início ou acrescentá-lo no fim.

15.7.5. Zend_Form_Decorator_Fieldset

Grupos de exibição e subformulários renderizam seu conteúdo de dentro de conjuntos de campo por padrão. O decorator FieldSet verifica a existência de uma opção 'legend' ou um método `getLegend()` no elemento registrado, e use como uma legenda se não estiver vazia. Qualquer conteúdo passado envolvido no conjunto de de campos HTML, substituindo o conteúdo original. Quaisquer atributos configurados no item são passados como atributos HTML.

15.7.6. Zend_Form_Decorator_Form

Objetos `Zend_Form` tipicamente precisam renderizar uma tag de formulário HTML. O decorator Form substitui o helper view do Form. Ele envolve qualque contéúdo fornecido em um elemento de formulário HTML, usando actions e methods do objeto `Zend_Form` e quaisquer atributos como atributos HTML.

15.7.7. Zend_Form_Decorator_FormElements

Formulários, grupos de exibição, e subformulários são coleções de elementos. De modo a renderizar esses elementos, eles utilizam o decorator FormElements, que itera atarvés de todos os itens, chamando `render()` sobre cada um e juntando-os com o separador registrado. Você pode adicionar ou incluir o conteúdo passado no início.

15.7.8. Zend_Form_Decorator_HtmlTag

O decorator HtmlTag permite a você utilizar tags HTML para decorar conteúdo; a tag utilizada é passada na opção 'tag', e quaisquer outras opções são usadas como atributos HTML para aquela tag. A tag por padrão pe assumida para ser nívle de bloco, e substitui o conteúdo pelo envolvimento do mesmo na tag dada. Entretanto, você pode especificar uma colocação para adicionar ou incluir no início da tag.

15.7.9. Zend_Form_Decorator_Image

O decorattor Image permite a você criar uma imagem de entrada HTML (<input type="image" ... />), e opcionalmente renderizá-la de dentro de outra tag HTML.

Por padrão, o decorator usa a propriedade de elemento src, que pode ser configurada com o método `setImage()`, como a fonte de imagem. Adicionamente, o rótulo do elemento será usado como tag alt, e `imageValue` (manipulado com os acessores de elemento Image `setImageValue()` e `getImageValue()`) serão usados para o valor.

Para especificar uma tag HTML com a qual envolver o elemento, passe a opção 'tag' para o decorator, ou explicitamente chame `setTag()`.

15.7.10. Zend_Form_Decorator_Label

Elementos de formulário tipicamente tem rótulos, e o decorator Label é usado para renderizar esses rótulos. Ele substitui o view helper FormLabel e puxa o elemento rótulo usando o método `getLabel()` do elemento. Se nenhum rótulo estiver presente, nenhum é renderizado. Por padrão, rótulos são traduzidos quando um adaptador de tradução existe e uma tradução para o rótulo existe.

Você pode opcionalmente especificar uma opção 'tag'; se fornecida, ela envolve o rótulo naquela tag nível de bloco. Se a opção 'tag' estiver presente, e nenhum rótulo presente, a tag é renderizada com nenhum contéudo. Você pode especificar a classe para usar com a tag com a opção 'class' ou chamando `setClass()`.

Adicionalmente, você pode especificar prefixos e sufixos para usar quando exibir o elemento, baseado se o rótulo é ou não para um elemento opcional ou requerido. Casos de uso comum seriam adicionar um ':' ao rótulo, ou um '*' indicando que um item é requerido. Você pode fazer isso com as seguintes opções e métodos:

- `optionalPrefix`: configura o texto para prefixar o rótulo quando o elemento for opcional. Usa os acessores `setOptionalPrefix()` e `getOptionalPrefix()` para manipulá-lo.
- `optionalSuffix`: configura o texto para adicionar ao rótulo quando o elemento for opcional. Usa os acessores `setOptionalSuffix()` e `getOptionalSuffix()` para manipulá-lo.
- `requiredPrefix`: configura o texto para prefixar o rótulo quando o elemento for requerido. Usa os acessores `setRequiredPrefix()` e `getRequiredPrefix()` para manipulá-lo.
- `requiredSuffix`: configura o texto para ser adicionado ao rótulo quando o elemento for requerido. Usa os acessores `setRequiredSuffix()` e `getRequiredSuffix()` para manipulá-lo.

Por padrão, o decorator Label inclui no início do conteúdo fornecido; especifique uma opção 'placement' de 'append' para colocá-lo depois do conteúdo.

15.7.11. Zend_Form_Decorator_ViewHelper

A maioria dos elementos utilizam helpers `Zend_View` para renderizar, e isso é feito com o decorator ViewHelper. Com isso, você pode especificar uma tag 'helper' para configurar explicitamente o view helper a ser utilizado; se nenhum for fornecido, use o último segmento do nome de classe do elemento para determinar o helper, incluindo no início a string 'form': por exemplo, 'Zend_Form_Element_Text' procuraria por um view helper de 'formText'.

Quaisquer atributos do elemento fornecido são passados para o view helper como atributos do elemento.

Por padrão, esse decorator adiciona conteúdo, use a opção 'placement' para especificar colocação alternativa.

15.7.12. Zend_Form_Decorator_ViewScript

Algumas vezes você pode querer usar um view script para criar seus elementos; desse modo você pode ter controle bem granulado sobre seus elementos, passando o view script para um designer, ou simplesmente criando um modo de sobrescrever facilmente configuração baseado em qual módulo

você está esando (cada módulo poderia opcionalmente sobrescrever eview scripts de elemento para servir a suas próprias necessidades). O decorator ViewScript resolve esse problema.

O decorator ViewScript requer uma opção 'viewScript', fornecida para o decorator, ou como um atributo do elemento. Ele então renderiza o view script como um script parcial, significando que cada chamada para ele tem seu próprio escopo de variável; nenhuma variável de visão será injetada que não seja do próprio elemento. Várias variáveis são então populadas:

- `element`: o elemento sendo decorado
- `content`: o conteúdo passado para o decorator
- `decorator`: o próprio objeto decorator
- Adicionalmente, todas as opções passadas para o decorator via `setOptions()` que não são usadas internamente (tais placement, separator, etc.) são passadas para o view script como variáveis de visão.

Como um exemplo, você poderia ter o seguinte elemento:

```
<?php
// Configurar o decorator para o elemento para um simples, ViewScript,
decorator,
// especificando o viewScript como uma opção, e algumas opções extras:
$element->setDecorators(array(array('ViewScript', array(
    'viewScript' => '_element.phtml',
    'class'      => 'form element'
))));
// OU especificando o viewScript como um atributo de elemento:
$element->viewScript = '_element.phtml';
$element->setDecorators(array(array('ViewScript', array('class' => 'form element
'))));
?>
```

Você poderia então ter um view script como esse:

```
<div class="<?= $this->class ?>">
    <?= $this->formLabel($this->element->getName(), $this->element->getLabel())
?>
    <?= $this->{$this->element->helper}(
        $this->element->getName(),
        $this->element->getValue(),
        $this->element->getAttribs()
    ) ?>
    <?= $this->formErrors($this->element->getMessages()) ?>
    <div class="hint"><?= $this->element->getDescription() ?></div>
</div>
```

Substituindo conteúdo com um view script

Você pode achar útil para o view script substituir o conteúdo fornecido para o decorator – por exemplo, se você quer envolvê-lo. Você pode fazer isso especificando um valor booleano false para a opção 'placement' do decorator:

```
<?php
// Na criação do decorator:
$element->addDecorator('ViewScript', array('placement' => false));
```

```
// Aplicando para uma instância de decorator existente:
$decorator->setOption('placement', false);
// Aplicando para um decorator já anexado a um elemento:
$element->getDecorator('ViewScript')->setOption('placement', false);
// Dentro de um view script usado por um decorator:
$this->decorator->setOption('placement', false);
?>
```

Usar o decorator ViewScript é recomendado para quando você quiser ter um controle bem granulado sobre como seus elementos são renderizados.

15.8. Internationalização de Zend_Form

De modo crescente, desenvolvedores precisam adaptar seu conteúdo para múltiplas línguas e regiões. Zend_Form visa fazer tal tarefa trivial, e alavancar funcionalidades tanto em Zend_Translate quanto Zend_Validate para fazer isso.

Por padrão, nenhuma internacionalização (i18n) é executada. Para ligar as características de i18n em `Zend_Form`, você precisará instanciar um objeto `Zend_Translate` com um adaptador apropriado, e anexá-lo a `Zend_Form` e/ou `Zend_Validate`. Veja a documentação de Zend_Translate para mais informações sobre a criação do objeto e arquivos de tradução.

Tradução Pode Ser Desligada Por Item

Você pode desabilitar tradução para qualquer formulário, elemento, grupo de exibição, ou subformulário chamando seu método `setDisableTranslator($flag)` ou passando uma opção `disableTranslator` para o objeto. Isso pode ser útil quando você quer desativar seletivamente a tradução para elementos individuais ou conjuntos de elementos.

15.8.1. Inicializando I18n em Formulários

De modo a inicializar i18n em formulários, você irá precisar de um objeto `Zend_Translate` ou `Zend_Translate_Adapter`, como detalhado na documentação de `Zend_Translate`. Uma vez que você tenha um objeto de tradução, você tem várias opções:

- *A mais fácil:* adicioná-lo ao registro. Toda i18n atenta de componentes de Zend Framework autodescobrirá um objeto de tradução que está no registro debaixo da chave 'Zend_Translate' e usa-o para executar tradução e/ou localização:

  ```
  <?php
  // usa a chave 'Zend_Translate' key; $translate é um objeto
   Zend_Translate:
  Zend_Registry::set('Zend_Translate', $translate);
  ?>
  ```

  Isso será melhorado por `Zend_Form`, `Zend_Validate`, e `Zend_View_Helper_Translate`.

- Se toda a sua preocupação for sobre a tradução de mensagens de erro de validação, você pode registrar o objeto de tradução com `Zend_Validate_Abstract`:

  ```
  <?php
  ```

```
// Diz a todo classe de validação para usar um adaptador de tradução
específico:
Zend_Validate_Abstract::setDefaultTranslator($translate);
?>
```

- Alternativamente, você pode anexar ao objeto `Zend_Form` como um tradutor global. Isso tem o efeito de também traduzir mensagens de erro de validação:

```
<?php
// Diz a todas as classes para usar um adaptador de tradução específico,
assim como usar
// esse adaptador para traduzir mensagens de erro de validação:
Zend_Form::setDefaultTranslator($translate);
?>
```

- Finalmente, você pode anexar um tradutor para uma instância de formulário específica ou para elementos específicos usando seus métodos `setTranslator()`:

```
<?php
// Tell *this* form instance to use a specific translate adapter; it will
also
// be used to translate validation error messages for all elements:
$form->setTranslator($translate);
// Tell *this* element to use a specific translate adapter; it will also b
e used
// to translate validation error messages for this particular element:
$element->setTranslator($translate);
?>
```

15.8.2. Alvos I18N Padrão

Agora que você anexou um objeto de tradução, o que exatamente você pode traduzir por padrão?

- *Mensagens de erro de validação.* Mensagens de erro de validação podem ser traduzidas. Para fazer isso, use as várias constantes de código de erro das classes de validação `Zend_Validate` como os Ids de mensagem. Para mais informações sobre esses códigos, veja a documentação de Zend_Validate.
- *Rótulos.* Elementos rótulos serão traduzidos, se uma tradução existir.
- *Legendas de conjuntos de campo.* Grupos de exibição e subformulários renderizam em conjuntos de campo (fieldsets) por padrão. O decorator Fieldset tenta traduzir a legenda antes do conjunto de campos.
- *Descrições de Formulários e Elementos.* Todos os tipos de formulários (elemento, formulário, grupo de exibição, subformulário) permitem a especificação de um item opcional de descrição. O decorator Description pode ser usado para renderizar isso, e por padrão irá tomar o valor e tentar traduzí-lo.
- *Valores Multiopção.* Para os vários itens que herdam de `Zend_Form_Element_Multi` (incluindo o MultiCheckbox, Multiselect, e elementos Radio), os valores de opção (não as chaves) irão ser traduzidos se uma tradução estiver disponível; isso significa que os rótulos de opção apresentados para o usuário serão traduzidos.

- *Rótulos Submit e Button.* Os vários elementos Submit e Button (Button, Submit, e Reset) traduzirão o rótulo exibido para o usuário.

15.9. Uso Avançado de Zend_Form

`Zend_Form` tem uma fartura de funcionalidades, muitas das quais visam desenvolvedores experientes. Este capítulo visa documentar algumas dessas funcionalidades com exemplos e casos de uso.

15.9.1. Notação de Matriz

Muitos desenvolvedores web experimentados gostam de agrupar elementos de formulário relacionados usando notação de matriz nos nomes de elemento. Por exemplo, se você tem dois endereços que deseja capturar, um de entrega e um de cobrança, você pode ter elementos idênticos; agrupando-os em uma matriz, você pode garantir que eles sejam capturados separadamente. Tome o seguinte formulário como exemplo:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="recipient">Ship to:</label></dt>
            <dd><input name="recipient" type="text" value="" /></dd>
            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>
            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>
            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>
            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>
    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="payer">Bill To:</label></dt>
            <dd><input name="payer" type="text" value="" /></dd>
            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>
            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>
            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>
            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>
    <dl>
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>
        <dt></dt>
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

Neste exemplo, os endereços de entrega e cobrança contêm alguns campos idênticos, o que significa

que um sobrecerveria o outro. Nós podemos resolver isso usando notação de matriz:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="shipping-recipient">Ship to:</label></dt>
            <dd><input name="shipping[recipient]" id="shipping-recipient"
                type="text" value="" /></dd>
            <dt><label for="shipping-address">Address:</label></dt>
            <dd><input name="shipping[address]" id="shipping-address"
                type="text" value="" /></dd>
            <dt><label for="shipping-municipality">City:</label></dt>
            <dd><input name="shipping[municipality]" id="shipping-municipality"
                type="text" value="" /></dd>
            <dt><label for="shipping-province">State:</label></dt>
            <dd><input name="shipping[province]" id="shipping-province"
                type="text" value="" /></dd>
            <dt><label for="shipping-postal">Postal Code:</label></dt>
            <dd><input name="shipping[postal]" id="shipping-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>
    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="billing-payer">Bill To:</label></dt>
            <dd><input name="billing[payer]" id="billing-payer"
                type="text" value="" /></dd>
            <dt><label for="billing-address">Address:</label></dt>
            <dd><input name="billing[address]" id="billing-address"
                type="text" value="" /></dd>
            <dt><label for="billing-municipality">City:</label></dt>
            <dd><input name="billing[municipality]" id="billing-municipality"
                type="text" value="" /></dd>
            <dt><label for="billing-province">State:</label></dt>
            <dd><input name="billing[province]" id="billing-province"
                type="text" value="" /></dd>
            <dt><label for="billing-postal">Postal Code:</label></dt>
            <dd><input name="billing[postal]" id="billing-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>
    <dl>
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>
        <dt></dt>
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

No exemplo acima, nós conseguimos separar os endereços. No formulário submetido, nós agora temos três elementos, o elemento 'save' para o submit, e então duas matrizes, 'shipping' e 'billing', cada uma com chaves para seus vários elementos.

`Zend_Form` tenta automatizar esse processo com seus subformulários. Por padrão, subformulários renderizam a notação de matriz como mostrado na listagem de formulário HTML anterior, completa com ids. O nome da matriz é baseado no nome do subformulário, com as chaves baseadas nos elementos contidos no subformulário. Subformulários podem ser aninhados arbitrariamente de

forma profunda, e isso irá criar matrizes aninhadas para refletir a estrutura. Adicionalmente, as várias rotinas de validação em `Zend_Form` honram a estrutura de matriz, garantindo que seu formulário valida corretamente, não importan quão arbitrariamente profundo você aninhe seus subformulários. Você não precisa fazer coisa alguma para se beneficiar disso; esse comportamente está habilitado por padrão.

Adicionalmente, há facilidades que permitem a você mudar par anotação de matriz condicionalmente, assim como especificar a matriz para a qual um elemento ou coleção pertencem:

- `Zend_Form::setIsArray($flag)`: Configurando o marco para true, você pode indicar que um formulário inteiro deve ser tratado como uma matriz. Por padrão, o nome do formulário será usado como o nome da matriz, a menos que `setElementsBelongTo()` tenha sido chamado. Se o formulário não tem especificado o nome, ou se `setElementsBelongTo()` não tem sido configurado, esse marco será ignorado (assim como não há nome de matriz para o qual os elementos possam pertencer).

  Você pode determinar se um formulário está sendo tratado como uma matriz usando o acessor `isArray()`.

- `Zend_Form::setElementsBelongTo($array)`: Usando este método, você pode especificar o nome de uma matriz para a qual elementos do formulário pertencem. Você pode determinar o nome usando o acessor `getElementsBelongTo()`.

Adicionalmente, no nível do elemento, você pode especificar que elementos individuais podem pertencer a matrizes particulares usando o método `Zend_Form_Element::setBelongsTo()`. Entretanto, fazer isso pode causar problemas quando validar seu elemento, e não é recomendado na maioria dos casos. Contudo, você pode ocasionalmente querer saber a qual matriz um elemento pertence, o que você pode descobrir usando o acessor `getBelongsTo()`.

15.9.2. Multi-Page Forms

Atualmente, formulários multipáginas não são oficialmente suportados em `Zend_Form`; entretanto, a maior parte do suporte para implementá-los está disponível e pode ser utilizada com algumas ferramentas extras.

A chave para criar um formulário multipágina é utilizar subformulários, mas exibir somente um subformulário por página. Isso permite a você submeter um subformulário simples no momento e validá-lo, mas não processar o formulário até que todos os subformulários estejam completos.

Exemplo 15.7. Exemplo de Formulário de Registro

Usemos um formulário de registro como um exemplo. Para todos os propósitos, nós queremos capturar o nome e senha desejados em uma página inicial, então os metadados dos usuários – prenome, sobrenome, e localização – e finalmente permitir a eles decidir que listas de envio, se houver, eles gostariam de assinar.

Primeiro, criemos nosso próprio formulário, e definamos vários subformulários dentre dele:

```
<?php
class My_Form_Registration extends Zend_Form
{
    public function init()
    {
        // Cria subformulário de usuário: nome de usuário e senha
        $user = new Zend_Form_SubForm();
        $user->addElements(array(
```

```
    new Zend_Form_Element_Text('username', array(
        'required'   => true,
        'label'      => 'Username:',
        'filters'    => array('StringTrim', 'StringToLower'),
        'validators' => array(
            'Alnum',
            array('Regex', false, array('/^[a-z][a-z0-9]{2,}$/'))
        )
    )),
    new Zend_Form_Element_Password('password', array(
        'required'   => true,
        'label'      => 'Password:',
        'filters'    => array('StringTrim'),
        'validators' => array(
            'NotEmpty',
            array('StringLength', false, array(6))
        )
    )),
));
// Cria subformulários demográficos: prenome, sobrenome, e localização
$demog = new Zend_Form_SubForm();
$demog->addElements(array(
    new Zend_Form_Element_Text('givenName', array(
        'required'   => true,
        'label'      => 'Given (First) Name:',
        'filters'    => array('StringTrim'),
        'validators' => array(
            array('Regex', false, array('/^[a-z][a-z0-9., \'-]{2,}$/i'))
        )
    )),
    new Zend_Form_Element_Text('familyName', array(
        'required'   => true,
        'label'      => 'Family (Last) Name:',
        'filters'    => array('StringTrim'),
        'validators' => array(
            array('Regex', false, array('/^[a-z][a-z0-9., \'-]{2,}$/i'))
        )
    )),
    new Zend_Form_Element_Text('location', array(
        'required'   => true,
        'label'      => 'Your Location:',
        'filters'    => array('StringTrim'),
        'validators' => array(
            array('StringLength', false, array(2))
        )
    )),
));
// Cria subformulário de listas de envio
$listOptions = array(
    'none'        => 'No lists, please',
    'fw-general'  =>  'Zend Framework General List',
    'fw-mvc'      =>  'Zend Framework MVC List',
    'fw-auth'     =>  'Zend Framwork Authentication and ACL List',
    'fw-services' => 'Zend Framework Web Services List',
);
$lists = new Zend_Form_SubForm();
$lists->addElements(array(
    new Zend_Form_Element_MultiCheckbox('subscriptions', array(
        'label'        => 'Which lists would you like to subscribe to?',
        'multiOptions' => $listOptions,
        'required'     => true,
```

```
                'filters'      => array('StringTrim'),
                'validators'   => array(
                    array('InArray', false, array(array_keys($listOptions)))
                )
            )),
        ));
        // Anexa subformulários ao formulário principal
        $this->addSubForms(array(
            'user'  => $user,
            'demog' => $demog,
            'lists' => $lists
        ));
    }
}
```

Note que não há botões de submissão, e que nós não temos feito nada com os decorator dos subformulários – o que significa que por padrão eles serão exibidos como conjuntos de campos. Nós precisaremos ser capazes de sobrescrever isso assim como exibir cada subformulário individual, e adicionarem botões de submissão de modo que nós possamos realmente processá-los – o que também requer propriedes action e method. Adicionemos algum scaffolding[5] para nossa classe para fornecer essa informação:

```
class My_Form_Registration extends Zend_Form
{
    // ...
    /**
     * Prepare a sub form for display
     *
     * @param  string|Zend_Form_SubForm $spec
     * @return Zend_Form_SubForm
     */
    public function prepareSubForm($spec)
    {
        if (is_string($spec)) {
            $subForm = $this->{$spec};
        } elseif ($spec instanceof Zend_Form_SubForm) {
            $subForm = $spec;
        } else {
            throw new Exception('Invalid argument passed to ' . __FUNCTION__ . '
()');
        }
        $this->setSubFormDecorators($subForm)
             ->addSubmitButton($subForm)
             ->addSubFormActions($subForm);
        return $subForm;
    }
    /**
     * Add form decorators to an individual sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function setSubFormDecorators(Zend_Form_SubForm $subForm)
    {
        $subForm->setDecorators(array(
            'FormElements',
            array('HtmlTag', array('tag' => 'dl', 'class' => 'zend_form')),
```

5 Literalmente “andaime”, aqui usado no sentido de código minimamente suficiente para tornar a aplicação funcional

```
            'Form',
        ));
        return $this;
    }
    /**
     * Add a submit button to an individual sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function addSubmitButton(Zend_Form_SubForm $subForm)
    {
        $subForm->addElement(new Zend_Form_Element_Submit(
            'save',
            array(
                'label'    => 'Save and continue',
                'required' => false,
                'ignore'   => true,
            )
        ));
        return $this;
    }
    /**
     * Add action and method to sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function addSubFormActions(Zend_Form_SubForm $subForm)
    {
        $subForm->setAction('/registration/process')
                ->setMethod('post');
        return $this;
    }
}
```

Em seguida, nós adicionamos algum scaffolding em nosso action controller, e temos várias considerações. Primeira, nós precisamos nos certificar de que persistirmos dados de formulários entre requisições, de modo que nós possamos determinar quando abortar. Segunda, nós precisamos de alguma lógica para determinar quais segmentos de formulário já tem diso submetidos, e que subformulários exibir baseado nessa informação. Nós usaremos `Zend_Session_Namespace` para persistir dados, o que também ajudará a responder a questão de qual formulário submeter.

Criemos nosso controlador, e adicionemos um método para recuperar uma instância de formulário:

```
<?php
class RegistrationController extends Zend_Controller_Action
{
    protected $_form;
    public function getForm()
    {
        if (null === $this->_form) {
            require_once 'My/Form/Registration.php';
            $this->_form = new My_Form_Registration();
        }
        return $this->_form;
    }
}
```

Agora, adicionemos algumas funcionalidades para determinar qual formulário será exibido. Basicamente, até que o formulário inteiro seja considerado válido, nós precisamos continuar mostrando segmentos de formulário. Adicionalmente, nós queremos nos certificar que eles estão em uma ordem particular: usuário, demografia, e então listas. Nós podemos determinar que dado tem sido submetido verificando chaves particulares representando cada subformulário em nosso namespace de sessão.

```
class RegistrationController extends Zend_Controller_Action
{
    // ...
    protected $_namespace = 'RegistrationController';
    protected $_session;
    /**
     * Get the session namespace we're using
     *
     * @return Zend_Session_Namespace
     */
    public function getSessionNamespace()
    {
        if (null === $this->_session) {
            require_once 'Zend/Session/Namespace.php';
            $this->_session = new Zend_Session_Namespace($this->_namespace);
        }
        return $this->_session;
    }
    /**
     * Get a list of forms already stored in the session
     *
     * @return array
     */
    public function getStoredForms()
    {
        $stored = array();
        foreach ($this->getSessionNamespace() as $key => $value) {
            $stored[] = $key;
        }
        return $stored;
    }
    /**
     * Get list of all subforms available
     *
     * @return array
     */
    public function getPotentialForms()
    {
        return array_keys($this->getForm()->getSubForms());
    }
    /**
     * What sub form was submitted?
     *
     * @return false|Zend_Form_SubForm
     */
    public function getCurrentSubForm()
    {
        $request = $this->getRequest();
        if (!$request->isPost()) {
            return false;
        }
        foreach ($this->getPotentialForms() as $name) {
```

```
            if ($data = $request->getPost($name, false)) {
                if (is_array($data)) {
                    return $this->getForm()->getSubForm($name);
                    break;
                }
            }
        }
        return false;
    }
    /**
     * Get the next sub form to display
     *
     * @return Zend_Form_SubForm|false
     */
    public function getNextSubForm()
    {
        $storedForms    = $this->getStoredForms();
        $potentialForms = $this->getPotentialForms();
        foreach ($potentialForms as $name) {
            if (!in_array($name, $storedForms)) {
                return $this->getForm()->getSubForm($name);
            }
        }
        return false;
    }
}
```

Os métodos abaixo permitem que usemos notações tais como "`$subForm = $this->getCurrentSubForm();`" para recuperar o subformulário atual para validação, ou "`$next = $this->getNextSubForm();`" para obter o próximo a ser exibido.

Agora, demonstremos como processar e exibir os vários subformulários. Nós podemos usar `getCurrentSubForm()` para determinar se algum subformulário foi submetido (valores de retorno false indicam que nada foi exibido ou submetido), e `getNextSubForm()` para recuperar um formulário para ser exibido. Nós podemos então usar os métodos de formulário `prepareSubForm()` para garantir que o formulário está pronto para ser exibido.

Quando nós temos uma submissão de formulário, nós podemos validar o subformulário, e então verificar se o formulário inteiro etá válido agora. Para fazer essas tarefas, nós precisaremos de métodos adicionais que garantam que os dados submetidos foram adicionados à sessão, e que quando validar o formulário inteiro, nós validamos contra todos os segmentos da sessão:

```
<?php
class My_Form_Registration extends Zend_Form
{
    // ...

    /**
     * O subformulário é válido?
     *
     * @param  Zend_Form_SubForm $subForm
     * @param  array $data
     * @return bool
     */
    public function subFormIsValid(Zend_Form_SubForm $subForm, array $data)
    {
        $name = $subForm->getName();
        if ($subForm->isValid($data)) {
```

```
            $this->getSessionNamespace()->$name = $subForm->getValues();
            return true;
        }
        return false;
    }
    /**
     * O formulário inteiro é válido?
     *
     * @return bool
     */
    public function formIsValid()
    {
        $data = array();
        foreach ($this->getSessionNamespace() as $key => $info) {
            $data[$key] = $info;
        }
        return $this->getForm()->isValid($data);
    }
}
```

Agora que nós temos o trabalho externo fora do caminho, construamos ao métodos (actions) para este controlador. Nós precisaremos de uma página de chegada para o formulário, e então um método action 'process' para processar o formulário.

```
<?php
class My_Form_Registration extends Zend_Form
{
    // ...
    public function indexAction()
    {
        // Ou exibe a página atual novamente, ou pega o "próximo" (primeiro)
        // subformulário
        if (!$form = $this->getCurrentSubForm()) {
            $form = $this->getNextSubForm();
        }
        $this->view->form = $this->getForm()->prepareSubForm($form);
    }
    public function processAction()
    {
        if (!$form = $this->getCurrentSubForm()) {
            return $this->_forward('index');
        }
        if (!$this->subFormIsValid($form, $this->getRequest()->getPost())) {
            $this->view->form = $this->getForm()->prepareSubForm($form);
            return $this->render('index');
        }
        if (!$this->formIsValid()) {
            $form = $this->getNextSubForm();
            $this->view->form = $this->getForm()->prepareSubForm($form);
            return $this->render('index');
        }
        // Formulário válido!
        // Renderiza informações em uma página de verificação
        $this->view->info = $this->getSessionNamespace();
        $this->render('verification');
    }
}
```

Como você notará, o código real para processar o formulário é relativamente simples. Nós verificamos se temo uma submissão do subformulário atual, e se não, nós voltamos para a página de chegada. Se nós temos mesmo um subformulário, nós tentamos validá-lo, exibi-lo novamente se falhar. Se o subformulário for válido, nós então verificamos se o formulário é válido, o que indicaria que terminanos; se não, nós exibimos o próximo segmento de formulário. Finalmente, nós exibimos uma página de verificação com o conteúdo da sessão.

Os view scripts são muito simples:

```
<? // registration/index.phtml ?>
<h2>Registration</h2>
<?= $this->form ?>
<? // registration/verification.phtml ?>
<h2>Thank you for registering!</h2>
<p>
    Here is the information you provided:
</p>
<?
// Tem de fazer esse construtor de acordo como os itens são armazenados em
namespaces de sessão
foreach ($this->info as $info):
    foreach ($info as $form => $data): ?>
<h4><?= ucfirst($form) ?>:</h4>
<dl>
    <? foreach ($data as $key => $value): ?>
    <dt><?= ucfirst($key) ?></dt>
    <? if (is_array($value)):
        foreach ($value as $label => $val): ?>
    <dd><?= $val ?></dd>
        <? endforeach;
       else: ?>
    <dd><?= $this->escape($value) ?></dd>
    <? endif;
    endforeach; ?>
</dl>
<? endforeach;
endforeach ?>
```

Publicações vindouras de Zend Framework incluirão componentes para fazer formulários multipáginas mais simples pela abstração da sessão e lógica de ordenação. Nesse meio tempo, o exemplo acima deve servir como uma orientação razoável para executar essa tarefa em seu site.