



Chapter 3

Working with Forms

48 Zend Framework: A Beginner's Guide

Key Skills & Concepts

- Learn to programmatically create forms and form elements
- Understand how to filter and validate user input
- Protect your forms from Cross-Site Request Forgery (CSRF) attacks and spambots
- Control the appearance of form elements and error messages
- Create a working contact form

In the previous chapter, you learned how the Zend Framework implements the Model-View-Controller pattern, and you looked underneath the hood of the example application to see how it works. You also started to flesh out the example application by adopting a modular directory structure, adding a master layout, and creating custom controllers, views, and routes for static content.

Now, while you can certainly use the Zend Framework to serve up static content, doing so is a lot like using a bulldozer to knock over a tower of plastic blocks. There's nothing stopping you from doing it, but it's not really what the bulldozer was intended for, and you're liable to face hard questions about why there's a bulldozer in your living room in the first place! The Zend Framework is similar, in that it's intended to provide robust, elegant, and extensible solutions to complex Web application development tasks. The more complex the task, the better suited it is to the power and flexibility of the framework...and the more fun you'll have knocking it down!

In this chapter, you'll learn how the Zend Framework can simplify one of the most common application development tasks: creating Web forms and processing user input. You'll also apply this knowledge to add some interactivity to the SQUARE example application, by creating a contact form. So without further ado, let's jump right in!

Understanding Form Basics

To demonstrate how the Zend Framework can help you with forms, a brief yet illustrative example will suffice. If you're like most PHP developers, chances are that you've written a form-processing script like the following one at some point in your career:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
  </head>
```

```

<body>
  <h2>Create Item</h2>
  <?php
  if (!isset($_POST['submit'])) {
  // no POST submission, display form
  ?>
  <form method="post" action="/item/create">
    <table>
      <tr>
        <td>Item name:</td>
        <td><input type="text" name="name" size="30" /></td>
      </tr>

      <tr>
        <td>Item quantity:</td>
        <td><input type="text" name="qty" size="3" /></td>
      </tr>

      <tr>
        <td colspan="2">
          <input type="submit" name="submit" value="Submit" />
        </td>
      </tr>
    </table>
  </form>
  <?php
  } else {
  // POST submission, validate input
  if (trim($_POST['name']) == '') {
    die('ERROR: Missing value - Item name');
  }
  if (trim($_POST['qty']) == '') {
    die('ERROR: Missing value - Item quantity');
  }
  if ($_POST['qty'] <= 0) {
    die('ERROR: Invalid value - Item quantity');
  }

  // process input
  // eg: save to database
  // attempt a connection
  try {
    $pdo = new PDO('mysql:dbname=test;host=localhost', 'user',
'pass');

    // create and execute INSERT query
    $name = $pdo->quote($_POST['name']);
    $qty = $pdo->quote($_POST['qty']);

```

50 Zend Framework: A Beginner's Guide

```

        $sql = "INSERT INTO shoppinglist (name, qty) VALUES ($name,
$qty) ";
        $pdo->exec($sql) or die("ERROR: " . implode(":", $pdo-
>errorInfo()));

        // close connection
        unset($pdo);

        // display success message
        echo 'Thank you for your submission';
    } catch (Exception $e) {
        die("ERROR: " . $e->getMessage());
    }
}
?>
</body>
</html>

```

There's nothing very clever or complicated here. This script is divided into two parts, split by a conditional test that inspects the `$_POST` variable to determine if the form has been submitted. The first half displays an input form containing two fields and a submit button; the second half validates the input to ensure that it is in the correct format and then proceeds to escape it and insert it into a database.

Figure 3-1 illustrates what the form looks like.

Now, while the script and general approach that you've just seen work in practice, there's no denying that it has a couple of problems:

- The same script file contains both HTML interface elements and PHP business logic. As discussed in the previous chapter, this is both messy to look at and hard to maintain. It's also hard to enforce consistency between forms, since the code required to produce each form is customized to a very high degree.
- Every time you add a new field to the form in the first half of the script, you need to add a corresponding set of validation tests and error messages to the second half of the script. This is annoying, and often repetitive; witness that the first two tests in the previous example do essentially the same thing.
- There's no way to reuse validation tests from one form in other forms (unless you had the foresight to package them into classes or functions from the get-go). As a result, you often end up writing the same code time and time again, especially when working with forms that perform related or similar operations.

Create Item

Item name:

Item quantity:

Figure 3-1 A form created using standard HTML markup

The Zend Framework comes with a set of components, collectively referred to as Zend_Form, which addresses these problems. To illustrate, consider the following example, which uses Zend_Form to produce a result equivalent to the previous script:

```
<?php
class Form_Item_Create extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/item/create')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Item name:');
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->addValidator('NotEmpty', true)
            ->addValidator('Alpha', true)
            ->addFilter('HTMLEntities')
            ->addFilter('StringTrim');

        // create text input for quantity
        $qty = new Zend_Form_Element_Text('qty');
        $qty->setLabel('Item quantity:');
        $qty->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addValidator('NotEmpty', true)
            ->addValidator('Int', true)
            ->addFilter('HTMLEntities')
            ->addFilter('StringTrim');

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Submit')
            ->setOptions(array('class' => 'submit'));

        // attach elements to form
        $this->addElement($name)
            ->addElement($qty)
            ->addElement($submit);
    }
}

class ExampleController extends Zend_Controller_Action
{
```

52 Zend Framework: A Beginner's Guide

```

public function formAction()
{
    $form = new Form_Item_Create;
    $this->view->form = $form;
    if ($this->getRequest()->isPost()) {
        if ($form->isValid($this->getRequest()->getPost())) {
            $values = $form->getValues();
            $pdo = new PDO('mysql:dbname=test;host=localhost', 'user',
'pass');
            $sql = sprintf("INSERT INTO shoppinglist (name, qty)
                VALUES ('%s', '%d')", $values['name'], $values['qty']);
            $pdo->exec($sql);
            $this->_helper->getHelper('FlashMessenger')
                ->addMessage('Thank you for your submission');
            $this->_redirect('/index/success');
        }
    }
}

```

Figure 3-2 illustrates what the form looks like.

You'll immediately notice three things about the code that creates the form in Figure 3-2:

- There isn't a single line of HTML code in the script. Form and form elements are represented as PHP objects, and they are configured using object methods. This ensures consistency and produces a standards-compliant Web form.
- Predefined validators and filters are available for common input validation and sanitization tasks. This reduces the amount of work involved, produces more maintainable code, and avoids repetition. Validators can also be combined or extended to support custom requirements.
- Validators are specified at the same time as form fields. This allows the form to "know" what each field can support and to easily identify the source of input errors. A single field can also be associated with multiple validators for more stringent input validation.

It should be clear from these points that Zend_Form provides a convenient, maintainable, and extensible solution for input form creation and data validation. The remainder of this chapter will explore Zend_Form in detail, illustrating it in a practical context.

Create Item

Item name:

Item quantity:

Figure 3-2 A form created using the Zend_Form component

Creating Forms and Form Elements

From the previous section, you know that `Zend_Form` offers an object-oriented API for generating forms and validating user input. Under the `Zend_Form` approach, *forms* are represented as instances of, or objects inheriting from, the `Zend_Form` base class. This base class exposes a number of methods to control the operation of the form, including the `setAction()` method to set the form's action URL and the `setMethod()` method to set the submission method. There's also a catch-all `setAttribs()` method, which allows you to set other form attributes. Here's an example of using these methods:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/my/action')
            ->setAttribs(array(
                'class' => 'form',
                'id'    => 'example'
            ))
            ->setMethod('post');
    }
}
```

Form elements are added by instantiating objects of the corresponding `Zend_Form_Element_*` class, setting element properties via class methods, and then attaching them to the form with the `addElement()` method. Here's an example of adding a text input and a submit button to a form:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/my/action')
            ->setAttribs(array(
                'class' => 'form',
                'id'    => 'example'
            ))
            ->setMethod('post');

        // create text input for title
        $title = new Zend_Form_Element_Text('title');
        $title->setLabel('Title:')
            ->setOptions(array(
```

54 Zend Framework: A Beginner's Guide

```

        'size' => '35'
    ));

    // create submit button
    $submit = new Zend_Form_Element_Submit('submit', array(
        'label' => 'Submit',
        'class' => 'submit'
    ));

    // attach elements to form
    $this->addElement($title)
        ->addElement($submit);
    }
}

```

Element objects can be configured either by passing values to the object constructor or by using named object methods. In the previous example, the object constructor for the text input element was passed the element name in the constructor, and the `setLabel()` and `setOptions()` methods were then used to set the element label and display properties, respectively. On the other hand, the submit button was configured directly in the object constructor, which was passed an array of options as the second argument.

TIP

You can also attach descriptions to form fields with the `setDescription()` method.

If you prefer, you can also create form elements using the `createElement()` method, by passing the element type to the method as its first argument. Here's an example, which is equivalent to the previous one:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/my/action')
            ->setAttribs(array(
                'class' => 'form',
                'id'    => 'example'
            ))
            ->setMethod('post');

        // create text input for title
        $title = $this->createElement('text', 'title', array(
            'label' => 'Title:',
            'size'  => 35,
        ));
    }
}

```



```

// create submit button
$submit = $this->createElement('submit', 'submit', array(
    'label' => 'Submit',
    'class' => 'submit'
));

// attach elements to form
$this->addElement($title)
    ->addElement($submit);
}
}
}

```

TIP

In many of the code listings in this chapter, you'll see examples of *method chaining*, wherein one method appears to invoke another. This is an example of the Zend Framework's "fluent interface," which provides a convenient shortcut to configure form objects with minimal additional coding. The end result is also significantly more readable. You can read more about fluent interfaces in the links at the end of this chapter.

Working with Form Elements

By default, the Zend Framework ships with definitions for 16 form elements, ranging from simple text input elements to more complex multiple selection lists, and it's useful to learn more about them. Table 3-1 gives a list of these 16 elements, together with their corresponding class names.

The following sections examine these in more detail.

Text and Hidden Fields

Text input fields, password input fields, and larger text input areas are represented by the `Zend_Form_Element_Text`, `Zend_Form_Element_Password`, and `Zend_Form_Element_Textarea` classes, respectively, while hidden form fields are represented by the `Zend_Form_Element_Hidden` class. The following example demonstrates these elements in action:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
    }
}

```

56 Zend Framework: A Beginner's Guide

| Element Class | Description |
|---------------------------------|--|
| Zend_Form_Element_Text | Text input field |
| Zend_Form_Element_Hidden | Hidden field |
| Zend_Form_Element_Password | Password field |
| Zend_Form_Element_Radio | Radio button |
| Zend_Form_Element_Checkbox | Check box |
| Zend_Form_Element_MultiCheckbox | Group of related check boxes |
| Zend_Form_Element_Select | Selection list (single) |
| Zend_Form_Element_MultiSelect | Selection list (multiple) |
| Zend_Form_Element_Textarea | Text input field |
| Zend_Form_Element_File | File input field |
| Zend_Form_Element_Image | Image |
| Zend_Form_Element_Button | Button |
| Zend_Form_Element_Hash | Unique string (for session identification) |
| Zend_Form_Element_Captcha | CAPTCHA (for spam filtering) |
| Zend_Form_Element_Reset | Reset button |
| Zend_Form_Element_Submit | Submit button |

Table 3-1 Form Element Classes Included with the Zend Framework

```

$name = new Zend_Form_Element_Text('name');
$name->setLabel('First name:');
    ->setOptions(array('id' => 'fname'));

// create password input
$pass = new Zend_Form_Element_Password('pass');
$pass->setLabel('Password:');
    ->setOptions(array('id' => 'upass'));

// create hidden input
$uid = new Zend_Form_Element_Hidden('uid');
$uid->setValue('49');

// create text area for comments
$comment = new Zend_Form_Element_Textarea('comment');
$comment->setLabel('Comment:')
    ->setOptions(array(
        'id' => 'comment',

```

```

        'rows' => '10',
        'cols' => '30',
    ));

    // attach elements to form
    $this->addElement($name)
        ->addElement($pass)
        ->addElement($uid)
        ->addElement($comment);
}
}

```

Figure 3-3 illustrates the result.

Radio Buttons and Checkboxes

Radio buttons are represented by the `Zend_Form_Element_Radio` class, while checkboxes are represented by the `Zend_Form_Element_Checkbox` class. Here's an example of these two classes in action:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('id' => 'fname'));

        // create radio buttons for type
        $type = new Zend_Form_Element_Radio('type');
        $type->setLabel('Membership type:')
            ->setMultiOptions(array(
                'silver' => 'Silver',
                'gold' => 'Gold',
                'platinum' => 'Platinum'
            ))
            ->setOptions(array('id' => 'mtype'));

        // create checkbox for newsletter subscription
        $subscribe = new Zend_Form_Element_Checkbox('subscribe');
        $subscribe->setLabel('Subscribe to newsletter')

```

Example Form

First name:

Password:

Comment:

Figure 3-3 A form with text and hidden input elements

58 Zend Framework: A Beginner's Guide

```

        ->setCheckedValue('yes')
        ->setUncheckedValue('no');

    // attach elements to form
    $this->addElement($name)
        ->addElement($type)
        ->addElement($subscribe);
    }
}

```

The `setMultiOptions()` method of the `Zend_Form_Element_Radio` object accepts an array, and uses it to set the list of available radio button options. The keys of the array represent the form values that will be submitted, while the corresponding values represent the human-readable labels for each option. Similarly, the `setCheckedValue()` and `setUncheckedValue()` methods of the `Zend_Form_Element_Checkbox` object allow you to customize the value for the element's checked and unchecked states. By default, these values are set to 1 and 0, respectively.

Figure 3-4 illustrates the result.

If you'd like the user to select from a set of options, the `Zend_Form_Element_MultiCheckbox` class is often a better bet than the `Zend_Form_Element_Checkbox` class, because it exposes a `setMultiOptions()` method that allows for multiple items to be selected. The resulting collection is then formatted and submitted as an array. Here's an example of it in action:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('id' => 'fname'));

        // create radio buttons for type
        $type = new Zend_Form_Element_Radio('type');
        $type->setLabel('Pizza crust:')

```

Example Form

Name:

Membership type:

Silver

Gold

Platinum

Subscribe to newsletter

Figure 3-4 A form with radio buttons and check boxes

```

->setMultiOptions(array(
    'thin'    => 'Thin',
    'thick'   => 'Thick'
))
->setOptions(array('id' => 'type'));

// create checkbox for toppings
$toppings = new Zend_Form_Element_MultiCheckbox('toppings');
$toppings->setLabel('Pizza toppings:')
->setMultiOptions(array(
    'bacon'    => 'Bacon',
    'olives'   => 'Olives',
    'tomatoes' => 'Tomatoes',
    'pepperoni' => 'Pepperoni',
    'ham'      => 'Ham',
    'peppers'  => 'Red peppers',
    'xcheese'  => 'Extra cheese',
));

// attach elements to form
$this->addElement($name)
->addElement($type)
->addElement($toppings);
}
}
}

```

Figure 3-5 illustrates what the result looks like.

Selection Lists

Single- and multiple-selection lists are supported through the `Zend_Form_Element_Select` and `Zend_Form_Element_MultiSelect` classes. Like the `Zend_Form_Element_MultiCheckbox` class, they too expose a `setMultiOptions()` method that can be used to set up the list of available options. The following example demonstrates both these element types in action:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');
    }
}

```

Example Form

Name:

Pizza crust:

Thin

Thick

Pizza toppings:

Bacon

Olives

Tomatoes

Pepperoni

Ham

Red peppers

Extra cheese

Figure 3-5 A form with radio buttons and multiple check boxes

60 Zend Framework: A Beginner's Guide

```

// create text input for name
$name = new Zend_Form_Element_Text('name');
$name->setLabel('Name:');
    ->setOptions(array('id' => 'fname'));

// create selection list for source country
$from = new Zend_Form_Element_Select('from');
$from->setLabel('Travelling from:');
    ->setMultiOptions(array(
        'IN' => 'India',
        'US' => 'United States',
        'DE' => 'Germany',
        'FR' => 'France',
        'UK' => 'United Kingdom'
    ));

// create multi-select list for destination countries
$to = new Zend_Form_Element_MultiSelect('to');
$to->setLabel('Travelling to:');
    ->setMultiOptions(array(
        'IT' => 'Italy',
        'SG' => 'Singapore',
        'TR' => 'Turkey',
        'DK' => 'Denmark',
        'ES' => 'Spain',
        'PT' => 'Portugal',
        'RU' => 'Russia',
        'PL' => 'Poland'
    ));

// attach elements to form
$this->addElement($name)
    ->addElement($from)
    ->addElement($to);
}
}

```

Figure 3-6 illustrates the result.

File Upload Fields

If you're looking to upload one or more files through a form, you'll need the `Zend_Form_Element_File` class, which provides a browsable file input box. Here's an example of it in use:

```

<?php
class Form_Example extends Zend_Form
{

```

Example Form

Name:

Travelling from:

Travelling to:

Italy
Singapore
Turkey
Denmark
Spain
Portugal
Russia
Poland

Figure 3-6 A form with selection lists

```

public function init()
{
    // initialize form
    $this->setAction('/sandbox/example/form')
        ->setEnctype('multipart/form-data')
        ->setMethod('post');

    // create file input for photo upload
    $photo = new Zend_Form_Element_File('photo');
    $photo->setLabel('Photo:')
        ->setDestination('/tmp/upload');

    // attach elements to form
    $this->addElement($photo);
}
}

```

CAUTION

Remember that you must set the form encoding type to 'multipart/form-data' for form uploads to be correctly handled. This can be done using the `setEnctype()` method of the form object.

Figure 3-7 illustrates what it looks like.

Example Form

Photo:

Figure 3-7 A form with file input fields

TIP

If you're trying to upload multiple related files, there's a convenient `setMultiFile()` method that generates a sequence of file input fields and saves you the hassle of instantiating multiple `Zend_Form_Element_File` objects. You'll see an example of this in the next chapter.

Buttons

Every form needs a submit button, and some also need a reset button. These two critical form elements are represented by the `Zend_Form_Element_Submit` and `Zend_Form_Element_Reset` classes, respectively, and they're illustrated in the next listing:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for title

```

62 Zend Framework: A Beginner's Guide

```

$title = new Zend_Form_Element_Text('title');
$title->setLabel('Title:')
        ->setOptions(array('size' => '35'));

// create submit button
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Submit');

// create reset button
$reset = new Zend_Form_Element_Reset('reset');
$reset->setLabel('Cancel');

// attach elements to form
$this->addElement($title)
        ->addElement($submit)
        ->addElement($reset);
}
}

```

Figure 3-8 illustrates the resulting output.

If you're after a more generic form button, you'll find it in the `Zend_Form_Element_Button` class, which provides a simple, clickable form button that is useful for many different purposes. Image buttons can be generated with the `Zend_Form_Element_Image` class; use the `setImage()` method to specify the source image for the button. Here's an example of one such image button:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for title
        $title = new Zend_Form_Element_Text('title');
        $title->setLabel('Title:')
            ->setOptions(array('size' => '35'));

        // create image submit button
        $submit = new Zend_Form_Element_Image('submit');
        $submit->setImage('/images/submit.jpg');
    }
}

```

Example Form

Title:

Figure 3-8 A form with submit and reset buttons


```

    // attach elements to form
    $this->addElement($title)
        ->addElement($submit);
}
}
}

```

Figure 3-9 illustrates the resulting output.

Example Form

Title:



Figure 3-9 A form with an image button

Hash and CAPTCHA Fields

The Zend Framework includes two “special” form elements to assist in maintaining input security: the Hash and CAPTCHA elements. These are represented by the `Zend_Form_Element_Hash` and `Zend_Form_Element_Captcha` classes, respectively.

The Hash element uses a salt value to generate a unique key for the form and store it in the session. When the form is submitted, the hash value submitted with the form is automatically compared to the value stored in the session. If a match is found, the form submission is assumed to be genuine. If there is a mismatch, it’s a reasonable supposition that the form has been hijacked and is being used in a Cross-Site Request Forgery (CSRF) attack.

Here’s an example of using this element:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for number
        $cc = new Zend_Form_Element_Text('ccnum');
        $cc->setLabel('Credit card number:')
            ->setOptions(array('size' => '16'));

        // create text input for amount
        $amount = new Zend_Form_Element_Text('amount');
        $amount->setLabel('Payment amount:')
            ->setOptions(array('size' => '4'));

        // create hash
        $hash = new Zend_Form_Element_Hash('hash');
        $hash->setSalt('hf823hflw03j');

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Submit');

        // attach elements to form
    }
}

```

64 Zend Framework: A Beginner's Guide

```

        $this->addElement($cc)
            ->addElement($amount)
            ->addElement($hash)
            ->addElement($submit);
    }
}

```

The CAPTCHA element automatically generates a CAPTCHA verification input, which is a useful tool to filter out automated form submissions. More and more Web sites are using CAPTCHAs to reduce the number of false registrations and/or spam messages received through online forms. Although manually generating and verifying a CAPTCHA is a tedious process, the `Zend_Form_Element_Captcha` makes it as simple as adding a few lines of code to your form. Here's an example:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for user name
        $name = new Zend_Form_Element_Text('username');
        $name->setLabel('Username:')
            ->setOptions(array('size' => '16'));
    }
}

```

Ask the Expert

Q: What is a CSRF attack, and how do I protect against it?

A: Typically, when a user visits a protected Web site and validates his/her access credentials, a user session is generated and the access credentials are revalidated from the session data store on each request. A CSRF attack involves hijacking a validated user session and using the implicit trust relationship that already exists between the user and the host application to invisibly transmit unauthorized requests through input sources such as Web forms. By generating a unique hash value for each Web form and validating this value when the form is submitted, a developer is able to make it harder to perform this type of attack. Using a hash value also provides (limited) protection from automated spam mailers (“spambots”), and is more user-friendly than a CAPTCHA.

```

// create password input
$pass = new Zend_Form_Element_Password('password');
$pass->setLabel('Password:');
    ->setOptions(array('size' => '16'));

// create captcha
$captcha = new Zend_Form_Element_Captcha('captcha', array(
    'captcha' => array(
        'captcha' => 'Figlet',
        'wordLen' => 5,
        'timeout' => 300,
    )
));
$captcha->setLabel('Verification:');

// create submit button
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Sign Up');

// attach elements to form
$this->addElement($name)
    ->addElement($pass)
    ->addElement($captcha)
    ->addElement($submit);
}
}
}

```

Figure 3-10 illustrates what the result might look like.

NOTE

A number of predefined CAPTCHA adapters are included with the Zend Framework, including adapters for simple string-transposition operations (“Dumb”) and for visual CAPTCHAS (“Image” and “Figlet”). You’ll see another example of an image CAPTCHA a little further along in this chapter.

Setting Required and Default Values

You can mark a specific input element as required by calling its `setRequired()` method with a true argument. Here’s an example:

```

<?php
class Form_Example extends Zend_Form
{

    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');
    }
}

```

66 Zend Framework: A Beginner's Guide

Example Form

Username:

Password:

Verification:




Figure 3-10 A form containing a CAPTCHA

```
// create text input for name
$name = new Zend_Form_Element_Text('name');
$name->setLabel('Name:');
    ->setOptions(array('size' => '35'))
    ->setRequired(true);
```

Ask the Expert

Q: What is a CAPTCHA?

A: A CAPTCHA, or Completely Automated Public Turing test to tell Computers and Humans Apart, is a common challenge-response test used to identify whether the entity at the other end of a connection is a human being or a computer. On the Web, the typical form of a CAPTCHA is a distorted sequence of random alphanumeric characters, operating on the principle that a computer would be unable to see past the distortion, but a human, with greater powers of perception, would be able to correctly identify the sequence. Such CAPTCHAs are typically attached to input forms on the Web (for example, user registration forms), and they must be solved correctly before the input will be processed by the host application. CAPTCHAs need not always be visual; audio CAPTCHAs are also possible, and are most appropriate for visually handicapped users.

```

// create text input for email address
$email = new Zend_Form_Element_Text('email');
$email->setLabel('Email address:');
$email->setOptions(array('size' => '50'))
    ->setRequired(true);

// create submit button
$submit = new Zend_Form_Element_Submit('submit',
    array('class' => 'submit')
);
$submit->setLabel('Sign Up');

// attach elements to form
$this->addElement($name)
    ->addElement($email)
    ->addElement($submit);
}
}

```

When you use the `setRequired()` method on an input field, `Zend_Form` automatically attaches a `NotEmpty` validator to that field. As a result, if the field is empty when the form is submitted, an error message will appear. Figure 3-11 illustrates the result.

TIP

You can tell `Zend_Form` not to attach a `NotEmpty` validator to required elements by explicitly calling the element's `setAutoInsertNotEmptyValidator()` method with a false argument. Validators are discussed in detail in the next section.

You can attach default values to input elements by calling the element object's `setValue()` method with the default value or by calling the form object's `setDefaultValues()` method with an array of default values. For text input fields, this can be any string value; for

Example Form

Name:

- Value is required and can't be empty

Email address:

- Value is required and can't be empty

Figure 3-11 The result of submitting a form without required input values

68 Zend Framework: A Beginner's Guide

radio buttons and selection lists, it should be the index of the selected item. Here's an example, which demonstrates both of these methods:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:');
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->setValue('Enter your name');

        // create text input for email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:');
        $email->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->setValue('Enter your email address');

        // create radio buttons for type
        $type = new Zend_Form_Element_Radio('type');
        $type->setLabel('Membership type:');
            ->setMultiOptions(array(
                'silver' => 'Silver',
                'gold' => 'Gold',
                'platinum' => 'Platinum'
            ));

        // create checkbox for newsletter subscription
        $subscribe = new Zend_Form_Element_Checkbox('subscribe');
        $subscribe->setLabel('Subscribe to newsletter')
            ->setCheckedValue('yes')
            ->setUncheckedValue('no');

        // create selection list for source country
        $from = new Zend_Form_Element_Select('from');
        $from->setLabel('Country:');
            ->setMultiOptions(array(
                'IN' => 'India',
                'US' => 'United States',
                'DE' => 'Germany',
                'FR' => 'France',
```

```

        'UK' => 'United Kingdom'
    ));

    // create submit button
    $submit = new Zend_Form_Element_Submit(
        'submit', array('class' => 'submit'));
    $submit->setLabel('Sign Up');

    // attach elements to form
    $this->addElement($name)
        ->addElement($email)
        ->addElement($type)
        ->addElement($from)
        ->addElement($subscribe)
        ->addElement($submit);

    // set default values
    $this->setDefaults(array(
        'type' => 'platinum',
        'subscribe' => 'yes',
        'from' => 'FR',
    ));
}
}
}

```

Figure 3-12 illustrates what the result looks like.

Example Form

Name:

Email address:

Membership type:
 Silver
 Gold
 Platinum

Country:

Subscribe to newsletter

Figure 3-12 A form rendered with default values

Filtering and Validating Form Input

As a Web application developer, there's one unhappy fact that you'll have to learn to live with: There are always going to be people out there who get their chuckles from finding loopholes in your code and exploiting these loopholes for malicious purposes. Therefore, one of the most important things a developer can do to secure an application is to properly filter and validate all the input passing through it.

The following sections discuss the filtering and validation tools available in the Zend Framework, together with examples of how they can be used with Web forms to make your application more secure.

Using Input Filters

Most of the time, input exploits consist of sending your application cleverly disguised values that “trick” it into doing something it really, really shouldn't. A common example of this type of exploit is the SQL injection attack, wherein an attacker remotely manipulates your database with an SQL query embedded inside form input. Therefore, one of the most important things a developer must do before using any input supplied by the user is to “sanitize” it by removing any special characters or symbols from it.

PHP comes with various functions to assist developers in the task of sanitizing input. For example, the `addslashes()` function escapes special characters (like quotes and backslashes) in input so that it can be safely entered into a database, while the `strip_tags()` function strips all the HTML and PHP tags out of a string, returning only the ASCII content. There's also the `htmlentities()` function, which is commonly used to replace special characters like `"`, `&`, `<`, and `>` with their corresponding HTML entity values, rendering them harmless.

Here's an example of sanitizing form input with the `htmlentities()` function:

```
<?php
// define array of sanitized data
$sanitized = array();

// strip tags from POST input
if (isset($_POST['name']) && !empty($_POST['name'])) {
    $sanitized['name'] = htmlentities($_POST['name']);
}

// processing code //
?>
```

When it comes to filtering user input, the Zend Framework does a lot of the heavy lifting for you. The `Zend_Filter` component provides a comprehensive set of *input filters*, which can either be attached to form elements with the `addFilter()` method or used on a stand-alone basis for ad-hoc input sanitization. Here's an example of using the `HTMLEntities` filter on a text input field:

```
<?php
class Form_Example extends Zend_Form
```



```

{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for user name
        // filter special characters
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Username:')
            ->setOptions(array('size' => '16'))
            ->addFilter('HtmlEntities');

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // attach elements to form
        $this->addElement($name)
            ->addElement($submit);
    }
}

```

You can also pass the `addFilter()` method an instance of the `Zend_Filter_*` class, as shown in the following equivalent script:

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for user name
        // filter special characters
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Username:')
            ->setOptions(array('size' => '16'))
            ->addFilter(new Zend_Filter_HtmlEntities());

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // attach elements to form
    }
}

```

72 Zend Framework: A Beginner's Guide

```

        $this->addElement($name)
            ->addElement($submit);
    }
}

```

Some filters support additional options, which can be passed to the `addFilter()` method as an array or, if you're using a class instance, as arguments to the object constructor. Consider the next example, which uses the Alpha filter to strip out all non-alphabetic characters from user input. An additional option, passed to the `addFilter()` method as a second argument, retains whitespace (which is stripped by default).

```

<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        // allow alphabetic characters and whitespace
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addFilter('Alpha', array('allowWhiteSpace' => true))
            ->addFilter('HtmlEntities');

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // attach elements to form
        $this->addElement($name)
            ->addElement($submit);
    }
}

```

Table 3-2 gives a list of some important filters that ship with the Zend Framework, together with a brief description of each. You'll see many of these filters in use in this and subsequent chapters.

TIP

You can attach multiple filters to a form element in one of two ways: by calling the `addFilter()` method multiple times, with a different filter name on each invocation, or by using the `addFilters()` method and passing it an array containing a list of filter names.

| Filter Name | Description |
|-----------------------|--|
| Alnum | Removes non-alphanumeric characters from argument |
| Alpha | Removes non-alphabetic characters from argument |
| Digits | Removes non-numeric characters from argument |
| Int | Returns integer value of argument |
| Dir | Returns directory name component of argument |
| BaseName | Returns filename component of argument |
| RealPath | Returns absolute filesystem path for argument |
| StringToLower | Converts argument to a lowercase string |
| StringToUpper | Converts argument to an uppercase string |
| StringTrim | Removes leading and trailing whitespace from argument |
| StripNewlines | Removes line break characters from argument |
| HtmlEntities | Converts special characters in argument to their HTML entity equivalents |
| StripTags | Removes HTML and PHP code from argument |
| Encrypt | Returns encrypted version of argument |
| Decrypt | Returns decrypted version of argument |
| NormalizedToLocalized | Returns argument in standard form |
| LocalizedToNormalized | Returns argument in localized form |
| Callback | Calls user-defined filter with argument |
| LowerCase | Converts contents of uploaded file to lowercase |
| UpperCase | Converts contents of uploaded file to uppercase |
| Rename | Renames uploaded file |

Table 3-2 Input Filters Included with the Zend Framework

Using Input Validators

Filtering input is only part of the puzzle. It's also extremely important to validate user input to ensure that it is in the correct format before using it for calculations or saving it to the application's data store. Improperly validated application input can not only cause significant data corruption and loss, but it can also be embarrassing in the extreme to the proud application developer.

In order to illustrate the importance of input validation, consider a simple example: an online mortgage calculator that allows a user to enter the desired loan amount, finance term, and interest rate. Now, let's assume that the application doesn't include any input validation. And let's also suppose that the user decides to enter the string 'ten', instead of the number 10, into the term field.

74 Zend Framework: A Beginner's Guide

It shouldn't be too hard to guess what happens next. The application will perform a few internal calculations that will end in it attempt to divide the total amount payable by the specified term. Since the term in this case is a string, PHP will cast it to the number 0, producing a division-by-zero error. The resulting slew of ugly error messages is likely to leave even the most blasé developer red-faced; more importantly, if the invalid input is also saved to the database as is, the error will recur every time the calculation is repeated on the record. Multiply this by even a few hundred records containing similar errors, scattered throughout the database, and you'll quickly see how the lack of appropriate input validation can significantly damage an application.

PHP comes with various functions to assist developers in the task of validating input. For example, the `is_numeric()` function tests if a value is numeric, while the `ctype_alpha()` and `ctype_alnum()` functions can be used to test for alphabetic and alphanumeric strings. There's also the `filter_var()` function, which can be used to test the validity of email addresses and URLs, and the `preg_match()` function, which allows for pattern validation using regular expressions. Here's an example of some of these functions in action:

```
<?php
// define array of valid data
$valid = array();

// check if age is a number
if (is_numeric(trim($_POST['age']))) {
    $valid['age'] = trim($_POST['age']);
} else {
    die ('ERROR: Age is not a number.');
```

```
}

// check for valid first name
if (isset($_POST['firstname']) && ctype_alpha($_POST['firstname'])) {
    $valid['firstname'] = trim($_POST['firstname']);
} else {
    die ('ERROR: First name not present or invalid.');
```

```
}

// check for valid email address
if (isset($_POST['email'])
    && filter_var($_POST['email'], FILTER_VALIDATE_EMAIL)) {
    $valid['email'] = trim($_POST['email']);
} else {
    die ('ERROR: Email address not present or invalid.');
```

```
}

// processing code here //
?>
```

As with filters, the Zend Framework ships with a large number of predefined *input validators*, collectively referred to as `Zend_Validate`, which can either be attached to form elements with the `addValidator()` method or used ad hoc. Validator-specific options can

be passed as the third argument to the `addFilter()` method as an associative array of key-value pairs, as shown in the following example:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for age
        // should contain only integer values between 1 and 100
        $age = new Zend_Form_Element_Text('age');
        $age->setLabel('Age:')
            ->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addValidator('Int')
            ->addValidator('Between', false, array(1,100));

        // create text input for name
        // should contain only alphabetic characters and whitespace
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('First name:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator('Alpha', false, array('allowWhiteSpace' =>
true));

        // create text input for email address
        // should contain a valid email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator('EmailAddress');

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // attach elements to form
        $this->addElement($age)
            ->addElement($name)
            ->addElement($email)
            ->addElement($submit);
    }
}
```

76 Zend Framework: A Beginner's Guide

As with filters, validators can also be specified as instances of the corresponding `Zend_Validate_*` class, with validator options passed as arguments to the object constructor. The next example, which is equivalent to the previous one, illustrates this approach:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for age
        // should contain only integer values between 1 and 100
        $age = new Zend_Form_Element_Text('age');
        $age->setLabel('Age:')
            ->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_Int())
            ->addValidator(new Zend_Validate_Between(1,100));

        // create text input for name
        // should contain only alphabetic characters and whitespace
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('First name:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_Alpha(true));

        // create text input for email address
        // should contain a valid email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_EmailAddress());

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // attach elements to form
        $this->addElement($age)
            ->addElement($name)
            ->addElement($email)
            ->addElement($submit);
    }
}
```

Example Form

Age:

- '110' is not between '1' and '100', inclusively

First name:

- 'Vikram Vaswani The 1st' has not only alphabetic characters

Email address:

- 'example' is not a valid hostname for email address 'none@example'
- 'example' does not match the expected structure for a DNS hostname
- 'example' appears to be a local network name but local network names are not allowed

Figure 3-13 The result of submitting a form with invalid input values

Figure 3-13 illustrates the result of attempting to submit invalid values through such a form.

Table 3-3 provides a list of some important validators available in the Zend Framework, together with a brief description of each. You'll see many of these validators in use further along in this chapter, as well as in subsequent chapters.

| Validator Name | Description |
|----------------|--|
| NotEmpty | Returns false if argument is empty |
| StringLength | Returns false if argument does not conform to specified minimum/maximum length |
| InArray | Returns false if argument is not in specified array |
| Identical | Returns false if argument does not match specified value |
| Alnum | Returns false if argument does not contain only alphanumeric characters |
| Alpha | Returns false if argument does not contain only alphabetic characters |
| Int | Returns false if argument is not an integer |
| Float | Returns false if argument is not a floating-point number |
| Hex | Returns false if argument is not a hexadecimal value |
| Digits | Returns false if argument does not contain only numbers |
| Between | Returns false if argument is not in a specified numeric range |

Table 3-3 Input Validators Included with the Zend Framework

78 Zend Framework: A Beginner's Guide

| Validator Name | Description |
|------------------------|--|
| GreaterThan | Returns false if argument is not greater than a specified value |
| LessThan | Returns false if argument is not less than a specified value |
| Date | Returns false if argument is not a valid date |
| EmailAddress | Returns false if argument does not conform to standard email address conventions |
| Hostname | Returns false if argument does not conform to standard host name conventions |
| Ip | Returns false if argument does not conform to standard IP address conventions |
| Regex | Returns false if argument does not conform to specified regular expression pattern |
| Barcode | Returns false if argument is not a valid bar code |
| Ccnum | Returns false if argument does not conform to the Luhn algorithm for standard credit card number conventions |
| Iban | Returns false if argument is not a valid IBAN number |
| Exists | Returns false if argument is not a valid file |
| Count | Returns false if number of uploaded files is outside the range specified in argument |
| Size | Returns false if uploaded file size is outside the range specified in argument |
| FileSize | Returns false if uploaded file size total is outside the range specified in argument |
| Extension | Returns false if uploaded file extension does not match those specified in argument |
| MimeType | Returns false if uploaded file MIME type does not match those specified in argument |
| IsCompressed | Returns false if uploaded file is not a compressed archive file |
| IsImage | Returns false if uploaded file is not an image file |
| ImageSize | Returns false if uploaded image dimensions are outside the range specified in argument |
| Crc32, Md5, Sha1, Hash | Returns false if uploaded file content does not match the hash value specified in argument (supports crc32, md5, and sha1 hash algorithms) |
| ExcludeExtension | Returns false if uploaded file extension matches those specified in argument |
| ExcludeMimeType | Returns false if uploaded file MIME type matches those specified in argument |
| WordCount | Returns false if number of words in uploaded file is outside the range specified in argument |
| Db_RecordExists | Returns false if a particular record does not exist in the database and table specified in argument |
| Db_NoRecordExists | Returns false if a particular record exists in the database and table specified in argument |

Table 3-3 Input Validators Included with the Zend Framework (*continued*)

Ask the Expert

Q: I'm already validating form input using JavaScript. Why do I also need to validate it using PHP?

A: It's common practice to use client-side scripting languages like JavaScript or VBScript for client-side input validation. However, this type of client-side validation is not foolproof—if a user turns off JavaScript in the client, all your client-side code will become nonfunctional. That's why it's a good idea to couple client-side validation (which is faster) with server-side validation (which is more secure).

Using Validator and Filter Chains

One of the most interesting things about the `Zend_Filter` and `Zend_Validate` components is their support for *chaining* or *stacking*. Essentially, this means that it is possible to attach multiple filters and validators to a single input element, and have them automatically run, in sequence, once the form is submitted. The following example illustrates this by setting up a chain of four filters:

```
<?php
    // create text input for name
    // filter tags, entities and whitespace
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('First name:');
        ->setOptions(array('size' => '16'))
        ->setRequired(true)
        ->addFilter('StripTags')
        ->addFilter('HTMLEntities')
        ->addFilter('StringTrim')
        ->addFilter('StringToLower');
?>
```

In this example, the first filter strips HTML and PHP tags from the input, the second encodes entities, the third trims leading and trailing whitespace, and the fourth transforms the result to lowercase. These filters are executed on the input value in their order of appearance in the chain.

Validator chains work in a similar manner and come with an additional property. A validator chain can be configured such that a failure in any one validator terminates the entire chain with an error message. This behavior is controlled by the second argument to the `addValidator()` method which, when set to true, breaks the chain if there is a failure in the corresponding validator. Consider the next example, which illustrates this:

```
<?php
    // create text input for age
```

80 Zend Framework: A Beginner's Guide

```

// should contain only integer values between 1 and 100
$age = new Zend_Form_Element_Text('age');
$age->setLabel('Age:');
    ->setOptions(array('size' => '4'))
    ->setRequired(true)
    ->addValidator('NotEmpty', true)
    ->addValidator('Int', true)
    ->addValidator('Between', true, array(1,100));
?>

```

In this example, a failure in any one of the validators breaks the chain, and the remaining validators will not be processed. So, for example, if the input is not an integer value, the validation chain will terminate with the error message generated by the `Int` validator, and the `Between` validator will not be executed. Contrast this with the next listing:

```

<?php
// create text input for age
// should contain only integer values between 1 and 100
$age = new Zend_Form_Element_Text('age');
$age->setLabel('Age:');
    ->setOptions(array('size' => '4'))
    ->setRequired(true)
    ->addValidator('NotEmpty', false)
    ->addValidator('Int', false)
    ->addValidator('Between', false, array(1,100));
?>

```

In this version, even if one of the validators fails, the remaining validators will still be run, and error messages generated by any subsequent failures will be added to the message stack. This is illustrated in Figures 3-14 and 3-15, which compare and contrast the difference in behavior of these two listings.

Example Form

Age:

- Invalid type given, value should be a string or an integer

First name:

Email address:

Figure 3-14 A validator chain, broken on the first failure

Example Form

Age:

- Invalid type given, value should be a string or an integer
- '191.7' is not between '1' and '100', inclusively

First name:

Email address:

Figure 3-15 A validator chain, processed without any break

TIP

In case the predefined filters and validators that ship with the Zend Framework don't meet your needs, remember that you can always write your own. The Zend Framework manual has examples of how to do this.

Retrieving and Processing Form Input

Within a controller script, you can use a number of `Zend_Form` methods to retrieve and process form input after submission:

- The `isValid()` method checks if the submitted input is valid. This method accepts an array of input values and returns Boolean true or false depending on whether these values match the validation rules set up with the various `addValidator()` calls.
- If the input is invalid, the `getMessages()` method returns a list of the error messages generated during the validation process. This list can be processed and displayed when the form is re-rendered to give the user a hint about what went wrong.
- If the input is valid, the `getValues()` method can be used to retrieve the valid, filtered values for further processing. Input values are returned as elements of an associative array, where the array key represents the element name and the array value represents the corresponding input value. There's also a `getUnfilteredValues()` method, which returns the original, unfiltered input as entered by the user.

TIP

The `isValid()` method automatically verifies CAPTCHA and hash values, with no additional programming required on your part.

82 Zend Framework: A Beginner's Guide

The next listing illustrates how these methods are typically used in the context of a controller script:

```
<?php
class ExampleController extends Zend_Controller_Action
{
    public function formAction()
    {
        $form = new Form_Example;
        $this->view->form = $form;

        // check the request
        // run the validators
        if ($this->getRequest()->isPost()) {
            if ($form->isValid($this->getRequest()->getPost())) {
                // valid data: get the filtered and valid values
                // do something, save to database or write to file
                // display a success view
                $values = $form->getValues();
                $this->_redirect('/form/success');
            } else {
                // invalid data: get the error message array
                // for manual processing (if needed)
                // redisplay the form with errors
                $this->view->messages = $form->getMessages();
            }
        }
    }
}
?>
```

Try This 3-1 Creating a Contact Form

With all this background information at hand, let's now look at how it plays out in the context of a practical application. The following section applies everything you've learned so far to create an email inquiry form for the SQUARE application. This form will invite the user to enter a message and, on submission, will format the input into an email message and send it to the site administrators for follow-up.

Defining the Form

To begin, let's consider the requirements of the input form. They aren't very complicated—all that's really needed are three fields for the user to enter his or her name, email address, and message. These values should be validated, particularly the email address, to ensure authenticity and thereby make it possible for administrators to respond to email inquiries. To filter out automated submissions and reduce the incidence of spam, it would also be nice to include a

visual CAPTCHA—something that's quite easy to do with `Zend_Form`, as illustrated earlier.

Here's an example of what the resulting form definition would look like:

```
<?php
class Square_Form_Contact extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/contact/index')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->addValidator('NotEmpty', true)
            ->addValidator('Alpha', true)
            ->addFilter('HTMLEntities')
            ->addFilter('StringTrim');

        // create text input for email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:');
        $email->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->addValidator('NotEmpty', true)
            ->addValidator('EmailAddress', true)
            ->addFilter('HTMLEntities')
            ->addFilter('StringToLower')
            ->addFilter('StringTrim');

        // create text input for message body
        $message = new Zend_Form_Element_Textarea('message');
        $message->setLabel('Message:')
            ->setOptions(array('rows' => '8', 'cols' => '40'))
            ->setRequired(true)
            ->addValidator('NotEmpty', true)
            ->addFilter('HTMLEntities')
            ->addFilter('StringTrim');

        // create captcha
        $captcha = new Zend_Form_Element_Captcha('captcha', array(
            'captcha' => array(
                'captcha' => 'Image',
```

(continued)

84 Zend Framework: A Beginner's Guide

```

        'wordLen' => 6,
        'timeout' => 300,
        'width'   => 300,
        'height'  => 100,
        'imgUrl'  => '/captcha',
        'imgDir'  => APPLICATION_PATH . '/../public/captcha',
        'font'    => APPLICATION_PATH .
                    '/../public/fonts/LiberationSansRegular.ttf',
    )
));
$captcha->setLabel('Verification code:');

// create submit button
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Send Message')
        ->setOptions(array('class' => 'submit'));

// attach elements to form
$this->addElement($name)
     ->addElement($email)
     ->addElement($message)
     ->addElement($captcha)
     ->addElement($submit);
}
}

```

You should already be familiar with most of the preceding code. The form contains two text input elements for the user's name and email address, one text area for the message body, and a CAPTCHA element for verification. Alpha and NotEmpty validators are attached to the name and message body fields, while an EmailAddress validator is used to check the submitted email address. All fields are filtered using the HTMLEntities validator, and the email address is additionally converted to lowercase with the StringToLower validator.

The options passed to the `Zend_Form_Element_Captcha` instance are also worth looking into. Unlike the example shown in an earlier section, this definition generates a more complex CAPTCHA by dynamically overlaying a random sequence of characters on a distressed background. This type of CAPTCHA is commonly used in Web forms to stymie automated bot submissions, many of which include optical character recognition (OCR) algorithms that can “read” characters overlaid on a clear background. The options passed to the object instance include the dimensions of the CAPTCHA image, the disk location to store the generated CAPTCHA, the number of characters in the CAPTCHA, and the font file to use for the text overlay.

CAUTION

If you're using copyrighted fonts that cannot be redistributed, you should move the `$APP_DIR/public/fonts/` directory to a location outside the server document root, such as `$APP_DIR/application/fonts/`, to ensure that the fonts are not publicly accessible through a Web browser. If you're doing this, remember to update the application code to reflect the new path as well.

You'll notice that the previous example makes use of a custom font, and it stores generated CAPTCHAs in a specified directory. Accordingly, also create the `$APP_DIR/public/captcha/` and `$APP_DIR/public/fonts/` directories and copy over the necessary assets to these locations. You'll find these assets in the code archive for this chapter, which can be downloaded from this book's companion Web site at <http://www.zf-beginners-guide.com/>.

NOTE

The font used for the CAPTCHA in this example is the Liberation Sans font, part of a collection of fonts released to the community under the GNU General Public License by RedHat Inc. in 2007. Users are free to use, modify, copy, and redistribute these fonts under the terms of the GNU GPL.

Using a Custom Namespace

The definition in the previous section uses a custom namespace, "Square," which is prefixed to the class name. This is a recommended practice for any custom objects or libraries that you may create for the application, as it helps avoid name collisions between your definitions and others that may exist in the application space. An added benefit is that if you register your custom namespace with the Zend Framework's autoloader and then locate your definitions correctly in the application directory structure, the Zend Framework will automatically find and load them as needed at run time.

With this in mind, save the class definition from the preceding code to `$APP_DIR/library/Square/Form/Contact.php`, and then add the following directive to the application configuration file, at `$APP_DIR/application/configs/application.ini`, to register the "Square" namespace with the autoloader:

```
autoloaderNamespaces [] = "Square_"
```

CAUTION

If your classes use an underscore to separate the namespace from the rest of the class name, you must include this underscore when registering the namespace with the Zend Framework autoloader.

Defining a Custom Route

This is also a good time to add a custom route for the new form. While you've got the application configuration file open in your text editor, add the following route definition to it:

```
resources.router.routes.contact.route = /contact
resources.router.routes.contact.defaults.module = default
resources.router.routes.contact.defaults.controller = contact
resources.router.routes.contact.defaults.action = index
```

(continued)

86 Zend Framework: A Beginner's Guide

Based on the material discussed in Chapter 2, this should be quite familiar to you—it sets up a route such that requests for the application URL `/contact` are handled by the “default” module’s `ContactController::indexAction`.

Defining Controllers and Views

The next step is to define the aforesaid `ContactController::indexAction`. By convention, this controller should be located at `$APP_DIR/application/modules/default/controllers/ContactController.php`, and should look something like this:

```
<?php
class ContactController extends Zend_Controller_Action
{

    public function init()
    {
        $this->view->doctype('XHTML1_STRICT');
    }

    public function indexAction()
    {
        $form = new Square_Form_Contact();
        $this->view->form = $form;
        if ($this->getRequest()->isPost()) {
            if ($form->isValid($this->getRequest()->getPost())) {
                $values = $form->getValues();
                $mail = new Zend_Mail();
                $mail->setBodyText($values['message']);
                $mail->setFrom($values['email'], $values['name']);
                $mail->addTo('info@square.example.com');
                $mail->setSubject('Contact form submission');
                $mail->send();
                $this->_helper->getHelper('FlashMessenger')
                    ->addMessage('Thank you. Your message was successfully
sent.');
```

```
                $this->_redirect('/contact/success');
```

```
            }
        }
    }

    public function successAction()
    {
        if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
            $this->view->messages =
                $this->_helper->getHelper('FlashMessenger')->getMessages();
        } else {
```



```

        $this->_redirect('/');
    }
}
}

```

Most of the heavy lifting here is done by the `indexAction()` method, which creates an object of the `Square_Form_Contact` class discussed earlier and attaches it to the view. When the form is submitted, the object's `isValid()` method is used to validate the input submitted by the user. If the input is found to be valid, an instance of the `Zend_Mail` component is created, and object methods are used to format the input into an email message and send it to a specified email address. Once the message has been sent, control is transferred to the `successAction()` method, which renders a success view.

That's what happens if all goes well...but there's many a slip 'twixt the cup and the lip, so it's useful to understand what happens if things go wrong. If the input is found to be invalid, the `isValid()` method will return false and the form will be redisplayed, with error messages indicating the source of the error(s). `Zend_Form` will also automatically populate the form with the original input values to ensure that the user doesn't need to re-enter all the requested data. On the other hand, if the input is valid but an error occurs in the process of email generation and transmission, `Zend_Mail` will throw a PHP exception, which will be caught and handled by the application's default error handler.

NOTE

In order for the `Zend_Mail` object's `send()` method to work correctly, a mail delivery agent (such as *sendmail*) must be available and correctly configured in your *php.ini* configuration file. If this is not the case, message transmission will fail and the `send()` method will throw an exception at the point of failure.

This controller also introduces a new tool, the `FlashMessenger` helper, which is a useful little "helper" to simplify the display of status messages to the user. Messages can be added to the `FlashMessenger` object via its `addMessage()` method; these messages are then stored in the session until retrieved with a call to the `getMessages()` method, at which point they are removed from the session. This makes the `FlashMessenger` a convenient place to temporarily store messages between the time an operation ends and the time the subsequent view completes rendering, and you'll see it being used frequently throughout this book.

CAUTION

You'll notice that the controller's `init()` method sets the view's document type to `XHTML 1.0 Strict`. This is because, by default, `Zend_Form` doesn't produce well-formed `XHTML` markup. Setting the document type in this manner forces it to do so.

Obviously, you also need a couple of views, one for the input form and one for the success message. Here's the input view, which by convention should be located at `$APP_DIR/application/modules/default/views/scripts/contact/index.phtml`:

```

<h2>Contact</h2>
<?php echo $this->form; ?>

```

(continued)

88 Zend Framework: A Beginner's Guide

And here's the success view, which by convention should be stored at `$APP_DIR/application/modules/default/views/scripts/contact/success.phtml`:

```
<h2>Success</h2>
<?php echo implode($this->messages); ?>
```

Updating the Master Layout

All that's left now is to update the navigation links in the application's main menu to reflect the new inquiry form using the `url()` helper method. To do this, update the master layout, at `$APP_DIR/application/layouts/master.phtml`, with the changes highlighted in bold:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"/>
    <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
  </head>
  <body>
    <div id="header">
      <div id="logo">
        
      </div>

      <div id="menu">
        <a href="<?php echo $this->url(array(), 'home'); ?>">HOME</a>
        <a href="<?php echo $this->url(array('page' => 'services'),
'static-content'); ?>">SERVICES</a>
        <a href="<?php echo $this->url(array(), 'contact');
?>">CONTACT</a>
      </div>
    </div>

    <div id="content">
      <?php echo $this->layout()->content ?>
    </div>

    <div id="footer">
      <p>Created with <a href="http://framework.zend.com/">
Zend Framework</a>. Licensed under
      <a href="http://www.creativecommons.org/">Creative Commons
      </a>.</p>
    </div>
  </body>
</html>
```



Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://square.localhost/contact

Contact

Name:

Email address:

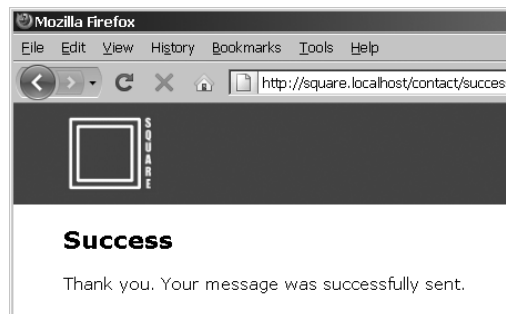
Message:

Verification code:


Figure 3-16 The SQUARE contact form

If you now try visiting the URL <http://square.localhost/contact> through your Web browser, you should see a form like the one in Figure 3-16.

Enter values into the form fields and submit it; if all goes well, you should see a success message like the one in Figure 3-17.



Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://square.localhost/contact/success

Success

Thank you. Your message was successfully sent.

Figure 3-17 The result of successfully submitting the SQUARE contact form

(continued)

90 Zend Framework: A Beginner's Guide

Contact

Name:


 'John The 2nd' has not only alphabetic characters

Email address:

 'none' is not a valid hostname for email address 'none@none'
 'none' does not match the expected structure for a DNS hostname
 'none' appears to be a local network name but local network names are not allowed

Message:

Value is required and can't be empty

Verification code:


 Captcha value is wrong

Figure 3-18 The result of submitting the SQUARE contact form with invalid input values

You can also try submitting the form with invalid values. The built-in validators will catch your invalid input and redisplay the form with error messages, as shown in Figure 3-18.

NOTE

Remember to update the recipient email address in the `ContactController::indexAction` to reflect your own email address, or else the email messages generated by the `Zend_Mail` component will never be received by you.

Customizing Form Appearance

You should now have a reasonably good idea about what goes into building a form with the Zend Framework, as well as some insight into the tools available to help you secure your application against invalid and malicious input. This section takes a quick look at some of the tools available to help you improve the appearance and behavior of your forms, with a view to making them clearer and more informative.

Using Custom Error Messages

Each input validator comes with a set of default error messages appropriate to the type of data being validated. More often than not, these default error messages provide enough information for users to locate and correct the errors in their input. However, cases may arise when these default messages need to be modified to be more descriptive and user-friendly.

This is not very difficult to do, because each validator can be configured to display custom error messages via the 'messages' key of the options array passed to the `addValidator()` method. Consider the following example, which illustrates this:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->addValidator('Alpha', false, array(
                'messages' => array(
                    Zend_Validate_Alpha::INVALID
                        => "ERROR: Invalid name",
                    Zend_Validate_Alpha::NOT_ALPHA
                        => "ERROR: Name cannot contain non-alpha characters",
                    Zend_Validate_Alpha::STRING_EMPTY
                        => "ERROR: Name cannot be empty"
                )
            ))
            ->addFilter('StringTrim');
        $validator = $name->getValidator('Alpha');

        // create text input for email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:');
        $email->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->addValidator('EmailAddress', true, array(
                'messages' => array(
                    Zend_Validate_EmailAddress::INVALID
                        => "ERROR: Invalid email address",
                    Zend_Validate_EmailAddress::INVALID_FORMAT
                        => "ERROR: Invalid email address",
```

92 Zend Framework: A Beginner's Guide

```

        Zend_Validate_EmailAddress::INVALID_HOSTNAME
            => "ERROR: Invalid hostname format",
        Zend_Validate_EmailAddress::INVALID_LOCAL_PART
            => "ERROR: Invalid username format",
        Zend_Validate_EmailAddress::LENGTH_EXCEEDED
            => "ERROR: Email address too long"
    )
))
->addFilter('StringTrim');

// create submit button
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Sign Up');

// attach elements to form
$this->addElement($name)
    ->addElement($email)
    ->addElement($submit);
}
}

```

In this example, the default error messages for each invalid case are overridden by the custom messages specified in the 'messages' key of the array passed to the `addValidator()` method. The constants on the left side of the message array can be obtained by inspecting the corresponding validator's source code. These new messages will then be generated, added to the error stack, and displayed in the form whenever the corresponding element fails validation. An example of the output is shown in Figure 3-19.

TIP

You can also pass custom error messages to a validator via its `setMessages()` method.

Example Form

Name:

- ERROR: Name cannot contain non-alpha characters

Email address:

- ERROR: Email address too long

Figure 3-19 A form with custom error messages

Using Display Groups

It's usually a good idea to group together form elements that have some tenuous or not-so-tenuous connection to each other. This is a more usable and readable approach than a collection of unordered elements clumped together without any clear categorization. With `Zend_Form`, this is accomplished through the use of *display groups*.

Display groups are added to a form with the form object's `addDisplayGroup()` method, which accepts two arguments: an array containing the names of the elements in the group, and a name for the group. The elements in question should have already been added to the form with the `addElement()` method. The `setLegend()` method can then be used to specify a name for the display group. At render time, these display groups and legends are represented by the `<fieldset>` and `<legend>` elements, respectively.

Here's an illustrative example:

```
<?php
class Form_Example extends Zend_Form
{
    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post');

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:');
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->addFilter('StringTrim');

        // create text input for email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:');
        $email->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->addValidator('EmailAddress', true)
            ->addFilter('StringTrim');

        // create text input for tel number
        $tel = new Zend_Form_Element_Text('tel');
        $tel->setLabel('Telephone number:');
        $tel->setOptions(array('size' => '50'))
            ->setRequired(true);

        // create submit button
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');
```

94 Zend Framework: A Beginner's Guide

```

// attach elements to form
$this->addElement($name)
    ->addElement($email)
    ->addElement($tel);

// add display group
$this->addDisplayGroup(
    array('name', 'email', 'tel', 'address'),
    'contact'
);
$this->getDisplayGroup('contact')
    ->setLegend('Contact Information');

$this->addElement($submit);
}
}

```

Figure 3-20 illustrates what the result looks like.

Using Decorators

When rendering a form, `Zend_Form` automatically wraps each form element in a set of HTML code blocks that control the location and appearance of the element. These blocks of HTML markup are referred to as *decorators* and, in addition to providing a ready hook for CSS-based manipulation, they can be customized on a per-element basis to radically alter how form elements are rendered.

Example Form

Figure 3-20 A form with display groups

| Decorator Name | Description |
|----------------|--|
| Form | Controls the markup around the form |
| FormElements | Controls the markup around form fields |
| HtmlTag | Controls the markup around form fields |
| ViewHelper | Controls the view helper |
| Errors | Controls the markup around validation errors (per field) |
| FormErrors | Controls the markup around validation errors (summary) |
| Description | Controls the markup around field descriptions |
| Label | Controls the markup around field labels |
| Fieldset | Controls the markup around fieldsets |

Table 3-4 Commonly Used Form Decorators

Like filters and validators, the Zend Framework ships with a number of default decorators. Table 3-4 lists the ones you're most likely to encounter.

The default Zend_Form decorators use the following markup:

- Form labels are wrapped in `<dt> . . . </dt>` elements.
- Input fields are wrapped in `<dd> . . . </dd>` elements.
- Validation errors are rendered as list items and wrapped in ` . . . ` elements.

Figure 3-21 illustrates the HTML source code of one such form, while Figure 3-22 illustrates the rendered result.

```
<html>
  <head>
  </head>
  <body>
    <h2>Example Form</h2>
    <form enctype="application/x-www-form-urlencoded" action="/sandbox/example/form" method="post"><dl class="z
<dt id="name-label"><label for="name" class="required">Name:</label></dt>
<dd id="name-element">
<input type="text" name="name" id="name" value="" size="35" />
<ul class="errors"><li>Value is required and can't be empty</li></ul></dd>
<dt id="email-label"><label for="email" class="required">Email address:</label></dt>
<dd id="email-element">
<input type="text" name="email" id="email" value="" size="50" />
<ul class="errors"><li>Value is required and can't be empty</li></ul></dd>
<dt id="subscribe-label"><label for="subscribe" class="optional">Subscribe to newsletter:</label></dt>
<dd id="subscribe-element">
<input type="hidden" name="subscribe" value="no" /><input type="checkbox" name="subscribe" id="subscribe" value
<dt id="submit-label">&nbsp;</dt><dd id="submit-element">
<input type="submit" name="submit" id="submit" value="Sign Up" class="submit" /></dd></dl></form> </body>
</html>
```

Figure 3-21 The source code of a form using default decorators

Example Form

Name:

- Value is required and can't be empty

Email address:

- Value is required and can't be empty

Subscribe to newsletter:

Figure 3-22 The rendered version of a form in Figure 3-21

Often, this arrangement is not suitable for your application's user interface. To alter it, use one of the following methods:

- The `addDecorators()` method accepts an array of decorator names and adds the corresponding decorators to the element. If you add a decorator that already exists, the decorator's settings will be overwritten with the values specified in the `addDecorators()` method.
- The `clearDecorators()` method removes all existing decorators for an element.
- The `setDecorators()` method accepts an array of decorator names, removes all existing decorators for an element, and attaches the new set of decorators to it.

TIP

To disable the default decorators for a particular form or form element, add a `'disableLoadDefaultDecorators'` key to that form or form element's options array. Note that if you're disabling the default decorators, you should still add back the `ViewHelper` decorator with the `addDecorators()` or `setDecorators()` method, as this decorator produces the basic markup for the form and its elements.

To illustrate how these decorators can be used, consider the following example:

```
<?php
class Form_Example extends Zend_Form
{
    public $formDecorators = array(
        array('FormElements'),
```

```
        array('Form'),
    );

    public $elementDecorators = array(
        array('ViewHelper'),
        array('Label'),
        array('Errors'),
    );

    public $buttonDecorators = array(
        array('ViewHelper'),
        array('HtmlTag', array('tag' => 'p'))
    );

    public function init()
    {
        // initialize form
        $this->setAction('/sandbox/example/form')
            ->setMethod('post')
            ->setDecorators($this->formDecorators);

        // create text input for name
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->setDecorators($this->elementDecorators);

        // create text input for email address
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:')
            ->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->setDecorators($this->elementDecorators);

        // create checkbox for newsletter subscription
        $subscribe = new Zend_Form_Element_Checkbox('subscribe');
        $subscribe->setLabel('Subscribe to newsletter:')
            ->setCheckedValue('yes')
            ->setUncheckedValue('no')
            ->setDecorators($this->elementDecorators);

        // create submit button
        $submit = new Zend_Form_Element_Submit(
            'submit', array('class' => 'submit'));
        $submit->setLabel('Sign Up')
            ->setDecorators($this->buttonDecorators);
    }
}
```

98 Zend Framework: A Beginner's Guide

```

    // attach elements to form
    $this->addElement($name)
        ->addElement($email)
        ->addElement($subscribe)
        ->addElement($submit);
    }
}

```

This listing resets the default decorators with the `setDecorators()` method, removing the `<dt>...</dt>` and `<dd>...</dd>` elements around labels and input fields and instead adjusting the appearance of these elements with CSS, which offers more precise control. Consider Figure 3-23, which illustrates the revised source code, and Figure 3-24, which illustrates the rendered result.

As this example illustrates, decorators provide a solution to the problem of controlling form markup such that it conforms to the rest of your application's interface. And in case you find that the default decorators don't offer enough control for you, remember that you can always define your own form markup by creating a custom decorator class (simply extend the `Zend_Form_Decorator_Abstract` abstract class) and using that instead. You'll find links to articles on this topic at the end of the chapter.

```

)
input (
    width: 300px;
)
input[type='checkbox'] {
    width: auto;
}
input.submit {
    width: 100px;
    margin-left: 210px;
}
label.required {
    color: red;
}
</style>
</head>
<body>
<h2>Example Form</h2>
<form enctype="application/x-www-form-urlencoded" action="/sandbox/example/form" method="post">
<label for="name" class="required">Name:</label>
<input type="text" name="name" id="name" value="" size="35" />
<ul class="errors"><li>Value is required and can't be empty</li></ul>
<label for="email" class="required">Email address:</label>
<input type="text" name="email" id="email" value="" size="50" />
<ul class="errors"><li>Value is required and can't be empty</li></ul>
<label for="subscribe" class="optional">Subscribe to newsletter:</label>
<input type="hidden" name="subscribe" value="no" /><input type="checkbox" name="subscribe" id="subscribe" value="yes" />
<p>
<input type="submit" name="submit" id="submit" value="Sign Up" class="submit" /></p></form> </body>
</html>

```

Figure 3-23 The source code of a form using custom decorators

Example Form

Name:

Value is required and can't be empty

Email address:

Value is required and can't be empty

Subscribe to newsletter:

Figure 3-24 The rendered version of the form in Figure 3-23

NOTE

Decorators are an extensive topic in themselves, and it isn't possible to cover them in exhaustive detail in this book. The material presented in the preceding section should be enough to get you started and to help you understand the relatively simple decorators used in this book. For more in-depth study, refer to the links at the end of this chapter.

Summary

This chapter focused on forms: creating them, validating them, and processing them. It introduced you to three of the most important components in the Zend Framework—`Zend_Form`, `Zend_Validate`, and `Zend_Filter`—and showed you how they can be used to securely request and handle user input. It then applied this knowledge to the SQUARE example application, guiding you through the process of registering a new namespace for custom objects, creating an email inquiry form, and transmitting the input received through this form as an email message using the `Zend_Mail` component. Finally, it took a quick look at some of the tools available to help you control the appearance of your forms, including display groups, decorators, and custom error messages; you'll see these tools in use throughout this book.

To learn more about the topics discussed in this chapter, consider visiting the following links:

- Wikipedia's discussion of CSRF and XSS attacks, at http://en.wikipedia.org/wiki/Cross-site_request_forgery and http://en.wikipedia.org/wiki/Cross-site_scripting
- Wikipedia's discussion of CAPTCHAs, at <http://en.wikipedia.org/wiki/Captcha>

100 Zend Framework: A Beginner's Guide

- Wikipedia's discussion of fluent interfaces, at http://en.wikipedia.org/wiki/Fluent_interface
- The Zend Framework autoloader, at <http://framework.zend.com/manual/en/zend.loader.autoloader.html>
- The Zend_Form component, at <http://framework.zend.com/manual/en/zend.form.html>
- The Zend_Filter component, at <http://framework.zend.com/manual/en/zend.filter.html>
- The Zend_Validate component, at <http://framework.zend.com/manual/en/zend.validate.html>
- The Zend_Mail component, at <http://framework.zend.com/manual/en/zend.mail.html>
- A discussion of creating custom decorators (Matthew Weier O'Phinney), at <http://weierophinney.net/matthew/archives/213-From-the-inside-out-How-to-layer-decorators.html>