symfony Forms in Action

symfony 1.2

This PDF is brought to you by
SENSIOLABS

Version: forms_book-1.2-en-2009-02-13

Table of Contents

About the Author

Fabien Potencier is a serial entrepreneur. In 1998, right after graduation, Fabien founded his very first company with a fellow student. The company was a web agency focused on simplicity and Open-Source technologies, and was called Sensio. His acute technical knowledge and his endless curiosity won him the confidence of many French big corporate companies.

Fabien is also the creator and the lead developer of the symfony framework.

Today, Fabien spends most of his time as Sensio's CEO and as the symfony project leader.

About Sensio Labs

Sensio Labs is a French web agency well known for its innovative ideas on web development. Founded in 1998 by Fabien Potencier, Gregory Pascal, and Samuel Potencier, Sensio benefited from the Internet growth of the late 1990s and situated itself as a major player for building complex web applications. It survived the Internet bubble burst by applying professional and industrial methods to a business where most players seemed to reinvent the wheel for each project. Most of Sensio's clients are large French corporations, who hire its teams to deal with small- to middle-scale projects with strong time-to-market and innovation constraints.

Sensio Labs develops interactive web applications, both for dot-com and traditional companies. Sensio Labs also provides auditing, consulting, and training on Internet technologies and complex application deployment. It helps define the global Internet strategy of large-scale industrial players. Sensio Labs has projects in France and abroad.

For its own needs, Sensio Labs develops the symfony framework and sponsors its deployment as an Open-Source project. This means that symfony is built from experience and is employed in many web applications, including those of large corporations.

Since its beginnings ten years ago, Sensio has always based its strategy on strong technical expertise. The company focuses on Open-Source technologies, and as for dynamic scripting languages, Sensio offers developments in all LAMP platforms. Sensio acquired strong experience on the best frameworks using these languages, and often develops web applications in Django, Rails, and, of course, symfony.

Sensio Labs is always open to new business opportunities, so if you ever need help developing a web application, learning symfony, or evaluating a symfony development, feel free to contact us at fabien.potencier@sensio.com. The consultants, project managers, web designers, and developers of Sensio can handle projects from A to Z.

Chapter 1

Form Creation

A form is made of fields like hidden inputs, text inputs, select boxes, and checkboxes. This chapter introduces you to creating forms and managing form fields using the symfony forms framework.

Symfony 1.1 is required to follow the chapters of this book. You will also need to create a project and a `frontend` application to keep going. Please refer to the introduction for more information on symfony project creation.

Before we start

We will begin by adding a contact form to a symfony application.

Figure 1-1 shows the contact form as seen by users who want to send a message.

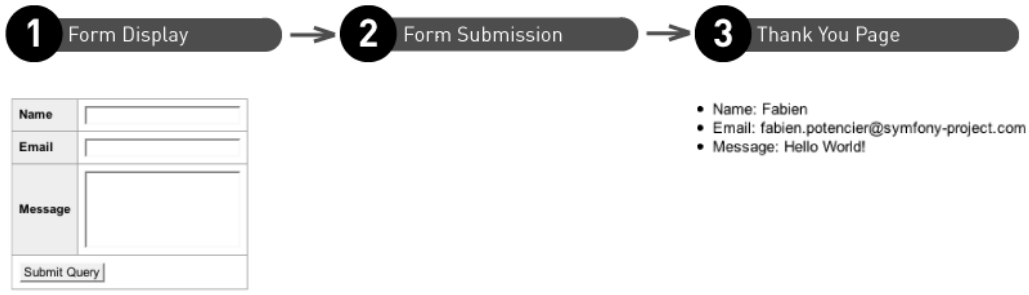
Figure 1-1 - Contact form

We will create three fields for this form: the name of the user, the email of the user, and the message the user wants to send. We will simply display the information submitted in the form for the purpose of this exercise as shown in Figure 1-2.

Figure 1-2 - Thank you Page

- Name: Fabien
- Email: fabien.potencier@symfony-project.com
- Message: Hello World!

Figure 1-3 - Interaction between the application and the user



Widgets

`sfForm` and `sfWidget` Classes

Users input information into fields which make up forms. In symfony, a form is an object inheriting from the `sfForm` class. In our example, we will create a `ContactForm` class inheriting from the `sfForm` class.

`sfForm` is the base class of all forms and makes it easy to manage the configuration and life cycle of your forms.

You can start configuring your form by adding **widgets** using the `configure()` method.

A **widget** represents a form field. For our form example, we need to add three widgets representing our three fields: `name`, `email`, and `message`. Listing 1-1 shows the first implementation of the `ContactForm` class.

Listing 1-1 - `ContactForm` class with three fields

Listing 1-1

```
// lib/form/ContactForm.class.php
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'message' => new sfWidgetFormTextarea(),
    ));
  }
}
```

The widgets are defined in the `configure()` method. This method is automatically called by the `sfForm` class constructor.

The `setWidgets()` method is used to define the widgets used in the form. The `setWidgets()` method accepts an associative array where the keys are the field names and the values are

the widget objects. Each widget is an object inheriting from the `sfWidget` class. For this example we used two types of widgets:

- `sfWidgetFormInput` : This widget represents the `input` field
- `sfWidgetFormTextarea`: This widget represents the `textarea` field

As a convention, we store the form classes in a `lib/form/` directory. You can store them in any directory managed by the symfony autoloading mechanism but as we will see later, symfony uses the `lib/form/` directory to generate forms from model objects.

Displaying the Form

Our form is now ready to be used. We can now create a symfony module to display the form:

Listing 1-2
```
$ cd ~/PATH/TO/THE/PROJECT
$ php symfony generate:module frontend contact
```

In the `contact` module, let's modify the `index` action to pass a form instance to the template as shown in Listing 1-2.

Listing 1-2 - Actions class from the `contact` *Module*

Listing 1-3
```
// apps/frontend/modules/contact/actions/actions.class.php
class contactActions extends sfActions
{
  public function executeIndex()
  {
    $this->form = new ContactForm();
  }
}
```

When creating a form, the `configure()` method, defined earlier, will be called automatically.

We just need to create a template now to display the form as shown in Listing 1-3.

Listing 1-3 - Template displaying the form

Listing 1-4
```
// apps/frontend/modules/contact/templates/indexSuccess.php
<form action="<?php echo url_for('contact/submit') ?>" method="POST">
  <table>
    <?php echo $form ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

A symfony form only handles widgets displaying information to users. In the `indexSuccess` template, the `<?php echo $form ?>` line only displays three fields. The other elements such as the `form` tag and the submit button will need to be added by the developer. This might not look obvious at first, but we will see later how useful and easy it is to embed forms.

Using the construction `<?php echo $form ?>` is very useful when creating prototypes and defining forms. It allows developers to concentrate on the business logic without worrying about visual aspects. Chapter three will explain how to personalize the template and form layout.

When displaying an object using the `<?php echo $form ?>`, the PHP engine will actually display the text representation of the `$form` object. To convert the object into a string, PHP tries to execute the magic method `__toString()`. Each widget implements this magic method to convert the object into HTML code. Calling `<?php echo $form ?>` is then equivalent to calling `<?php echo $form->__toString() ?>`.

We can now see the form in a browser (Figure 1-4) and check the result by typing the address of the action `contact/index` (`/frontend_dev.php/contact`).

Figure 1-4 - Generated Contact Form

Listing 1-4 Shows the generated code by the template.

Listing 1-5

```
<form action="/frontend_dev.php/contact/submit" method="POST">
  <table>

    <!-- Beginning of generated code by <?php echo $form ?> -->
    <tr>
      <th><label for="name">Name</label></th>
      <td><input type="text" name="name" id="name" /></td>
    </tr>
    <tr>
      <th><label for="email">Email</label></th>
      <td><input type="text" name="email" id="email" /></td>
    </tr>
    <tr>
      <th><label for="message">Message</label></th>
      <td><textarea rows="4" cols="30" name="message"
id="message"></textarea></td>
    </tr>
    <!-- End of generated code by <?php echo $form ?> -->

    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

We can see that the form is displayed with three `<tr>` lines of an HTML table. That is why we had to enclose it in a `<table>` tag. Each line includes a `<label>` tag and a form tag (`<input>` or `<textarea>`).

Labels

The labels of each field are automatically generated. By default, labels are a transformation of the field name following the two following rules: a capital first letter and underscores replaced by spaces. Example:

Listing 1-6
```
$this->setWidgets(array(
  'first_name' => new sfWidgetFormInput(), // generated label: "First name"
  'last_name'  => new sfWidgetFormInput(), // generated label: "Last name"
));
```

Even if the automatic generation of labels is very useful, the framework allows you to define personalized labels with the `setLabels()` method :

Listing 1-7
```
$this->widgetSchema->setLabels(array(
  'name'    => 'Your name',
  'email'   => 'Your email address',
  'message' => 'Your message',
));
```

You can also only modify a single label using the `setLabel()` method:

Listing 1-8
```
$this->widgetSchema->setLabel('email', 'Your email address');
```

Finally, we will see in Chapter three that you can extend labels from the template to further customize the form.

Widget Schema

When we use the `setWidgets()` method, symfony creates a `sfWidgetFormSchema` object. This object is a widget that allows you to represent a set of widgets. In our `ContactForm` form, we called the method `setWidgets()`. It is equivalent to the following code:

Listing 1-9
```
$this->setWidgetSchema(new sfWidgetFormSchema(array(
  'name'    => new sfWidgetFormInput(),
  'email'   => new sfWidgetFormInput(),
  'message' => new sfWidgetFormTextarea(),
)));

// almost equivalent to :

$this->widgetSchema = new sfWidgetFormSchema(array(
  'name'    => new sfWidgetFormInput(),
  'email'   => new sfWidgetFormInput(),
  'message' => new sfWidgetFormTextarea(),
));
```

The `setLabels()` method is applied to a collection of widgets included in the `widgetSchema` object .

We will see in the Chapter 5 that the "schema widget" notion makes it easier to manage embedded forms.

Beyond generated tables

Even if the form display is an HTML table by default, the layout format can be changed. These different types of layout formats are defined in classes inheriting from `sfWidgetFormSchemaFormatter`. By default, a form uses the `table` format as defined in the `sfWidgetFormSchemaFormatterTable` class. You can also use the `list` format:

Listing 1-10

```
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->widgetSchema->setFormFormatterName('list');
  }
}
```

Those two formats come by default and we will see in Chapter 5 how to create your own format classes. Now that we know how to display a form, let's see how to manage the submission.

Submitting the Form

When we created a template to display a form, we used the internal URL `contact/submit` in the `form` tag to submit the form. We now need to add the `submit` action in the `contact` module. Listing 1-5 shows how an action can get the information from the user and redirect to the `thank you` page where we just display this information back to the user.

Listing 1-5 - Use of the `submit` *action in the* `contact` *module*

Listing 1-11

```
public function executeSubmit($request)
{
  $this->forward404Unless($request->isMethod('post'));

  $params = array(
    'name'    => $request->getParameter('name'),
    'email'   => $request->getParameter('email'),
    'message' => $request->getParameter('message'),
  );

  $this->redirect('contact/thankyou?'.http_build_query($params));
}

public function executeThankyou()
{
}

// apps/frontend/modules/contact/templates/thankyouSuccess.php
<ul>
  <li>Name:    <?php echo $sf_params->get('name') ?></li>
  <li>Email:   <?php echo $sf_params->get('email') ?></li>
  <li>Message: <?php echo $sf_params->get('message') ?></li>
</ul>
```

`http_build_query` is a built-in PHP function that generates a URL-encoded query string from an array of parameters.

`executeSubmit()` method executes three actions:

- For security reasons, we check that the page has been submitted using the HTTP method `POST`. If not sent using the `POST` method then the user is redirected to a 404 page. In the `indexSuccess` template, we declared the submit method as `POST` (`<form ... method="POST">`):

Listing 1-12
```
$this->forward404Unless($request->isMethod('post'));
```

- Next we get the values from the user input to store them in the `params` table:

Listing 1-13
```
$params = array(
  'name'    => $request->getParameter('name'),
  'email'   => $request->getParameter('email'),
  'message' => $request->getParameter('message'),
);
```

- Finally, we redirect the user to a Thank you page (`contact/thankyou`) to display his information:

Listing 1-14
```
$this->redirect('contact/thankyou?'.http_build_query($params));
```

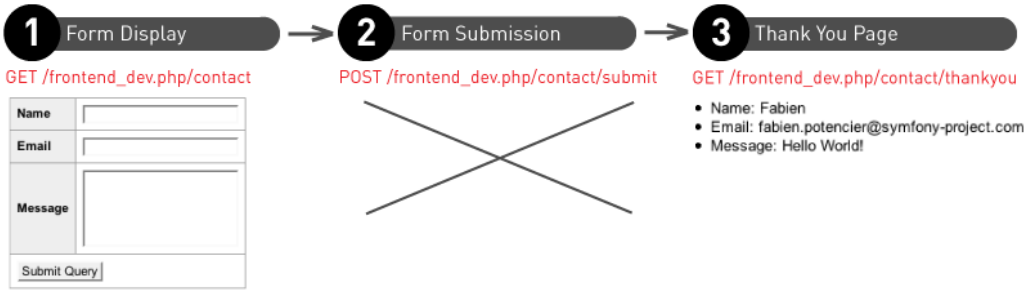
Instead of redirecting the user to another page, we could have created a `submitSuccess.php` template. While it is possible, it is better practice to always redirect the user after a request with the `POST` method:

- This prevents the form from being submitted again if the user reloads the Thank you page.
- The user can also click on the back button without getting the pop-up to submit the form again.

You might have noticed that `executeSubmit()` is different from `executeIndex()`. When calling these methods symfony passes the current `sfRequest` object as the first argument to the `executeXXX()` methods. With PHP, you do not have to collect all parameters, that is why we did not define the `request` variable in `executeIndex()` since we do not need it.

Figure 1-5 shows the workflow of methods when interacting with the user.

Figure 1-5 - Methods workflow

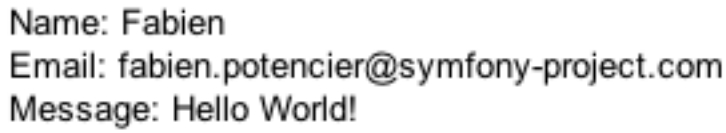


When redisplaying the user input in the template, we run the risk of a XSS (Cross-Site Scripting) attack. You can find further information on how to prevent the XSS risk by

> implementing an escaping strategy in the Inside the View Layer[1] chapter of "The Definitive Guide to symfony" book.

After you submit the form you should now see the page from Figure 1-6.

Figure 1-6 - Page displayed after submitting the form



Instead of creating the `params` array, it would be easier to get the information from the user directly in an array. Listing 1-6 modifies the `name` HTML attribute from widgets to store the field values in the `contact` array.

Listing 1-6 - Modification of the `name` *HTML attribute from widgets*

Listing 1-15

```
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->widgetSchema->setNameFormat('contact[%s]');
  }
}
```

Calling `setNameFormat()` allows us to modify the `name` HTML attribute for all widgets. `%s` will automatically be replaced by the name of the field when generating the form. For example, the `name` attribute will then be `contact[email]` for the `email` field. PHP automatically creates an array with the values of a request including a `contact[email]` format. This way the field values will be available in the `contact` array.

We can now directly get the `contact` array from the `request` object as shown in Listing 1-7.

Listing 1-7 - New format of the `name` *attributes in the action widgets*

Listing 1-16

```
public function executeSubmit($request)
{
  $this->forward404Unless($request->isMethod('post'));

  $this->redirect('contact/
thankyou?'.http_build_query($request->getParameter('contact')));
}
```

When displaying the HTML source of the form, you can see that symfony has generated a `name` attribute depending not only on the field name and format, but also an `id` attribute. The `id` attribute is automatically created from the `name` attribute by replacing the forbidden characters by underscores (_):

1. `http://www.symfony-project.org/book/1_1/07-Inside-the-View-Layer#Output%20Escaping`

Name	**Attribute** `name`	**Attribute** `id`
name	contact[name]	contact_name
email	contact[email]	contact_email
message	contact[message]	contact_message

Another solution

In this example, we used two actions to manage the form: `index` for the display, `submit` for the submit. Since the form is displayed with the `GET` method and submitted with the `POST` method, we can also merge the two methods in the `index` method as shown in Listing 1-8.

Listing 1-8 - Merging of the two actions used in the form

Listing 1-17
```
class contactActions extends sfActions
{
  public function executeIndex($request)
  {
    $this->form = new ContactForm();

    if ($request->isMethod('post'))
    {
      $this->redirect('contact/
thankyou?'.http_build_query($request->getParameter('contact')));
    }
  }
}
```

You also need to change the form `action` attribute in the `indexSuccess.php` template:

Listing 1-18
```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
```

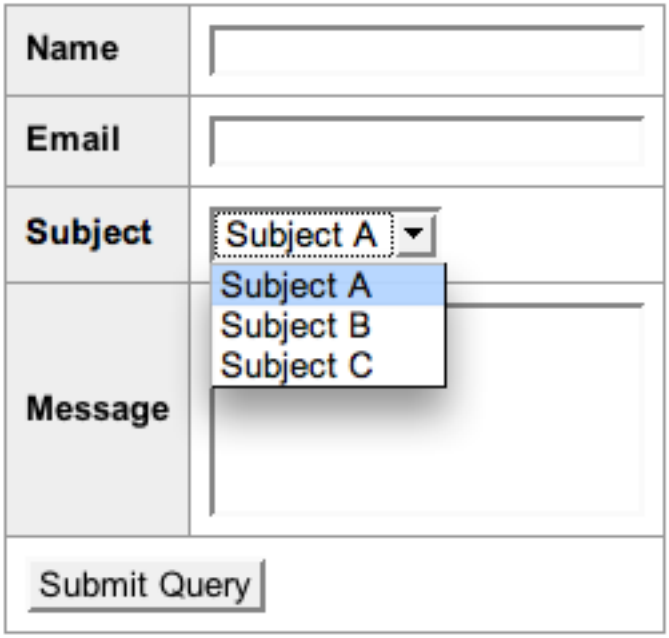
As we will see later, we prefer to use this syntax since it is shorter and makes the code more coherent and understandable.

Configuring the Widgets

Widgets options

If a website is managed by several webmasters, we would certainly like to add a drop-down list with themes in order to redirect the message according to what is asked (Figure 1-7). Listing 1-9 adds a `subject` with a drop-down list using the `sfWidgetFormSelect` widget.

Figure 1-7 - Adding a `subject` *Field to the Form*



Listing 1-9 - Adding a `subject` *Field to the Form*

Listing 1-19

```
class ContactForm extends sfForm
{
  protected static $subjects = array('Subject A', 'Subject B', 'Subject
C');

  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'subject' => new sfWidgetFormSelect(array('choices' =>
self::$subjects)),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->widgetSchema->setNameFormat('contact[%s]');
  }
}
```

The `choices` option of the `sfWidgetFormSelect` Widget

PHP does not make any distinction between an array and an associative array, so the array we used for the subject list is identical to the following code:

Listing 1-20
```
$subjects = array(0 => 'Subject A', 1 => 'Subject B', 2 => 'Subject C');
```

The generated widget takes the array key as the `value` attribute of the `option` tag, and the related value as content of the tag:

Listing 1-21
```
<select name="contact[subject]" id="contact_subject">
  <option value="0">Subject A</option>
  <option value="1">Subject B</option>
  <option value="2">Subject C</option>
</select>
```

In order to change the `value` attributes, we just have to define the array keys:

Listing 1-22
```
$subjects = array('A' => 'Subject A', 'B' => 'Subject B', 'C' => 'Subject
C');
```

Which generates the HTML template:

Listing 1-23
```
<select name="contact[subject]" id="contact_subject">
  <option value="A">Subject A</option>
  <option value="B">Subject B</option>
  <option value="C">Subject C</option>
</select>
```

The `sfWidgetFormSelect' widget, like all widgets, takes a list of options as the first argument. An option may be mandatory or optional. ThesfWidgetFormSelectwidget has a mandatory option,choices`. Here are the available options for the widgets we already used:

Widget	Mandatory Options	Additional Options
`sfWidgetFormInput`	-	`type` (default to `text`)
		`is_hidden` (default to `false`)
`sfWidgetFormSelect`	`choices`	`multiple` (default to `false`)
`sfWidgetFormTextarea`	-	-

If you want to know all of the options for a widget, you can refer to the complete API documentation available online at (http://www.symfony-project.org/api/1_1/[2]). All of the options are explained, as well as the additional options default values. For instance, all of the options for the `sfWidgetFormSelect` are available here: (http://www.symfony-project.org/api/1_1/sfWidgetFormSelect[3]).

2. `http://www.symfony-project.org/api/1_1/`
3. `http://www.symfony-project.org/api/1_1/sfWidgetFormSelect`

The Widgets HTML Attributes

Each widget also takes a list of HTML attributes as second optional argument. This is very helpful to define default HTML attributes for the generated form tag. Listing 1-10 shows how to add a `class` attribute to the `email` field.

Listing 1-10 - Defining Attributes for a Widget

```
$emailWidget = new sfWidgetFormInput(array(), array('class' => 'email'));

// Generated HTML
<input type="text" name="contact[email]" class="email" id="contact_email"
/>
```

Listing 1-24

HTML attributes also allow us to override the automatically generated identifier, as shown in Listing 1-11.

Listing 1-11 - Overriding the `id` Attribute

```
$emailWidget = new sfWidgetFormInput(array(), array('class' => 'email',
'id' => 'email'));

// Generated HTML
<input type="text" name="contact[email]" class="email" id="email" />
```

Listing 1-25

It is even possible to set default values to the fields using the `value` attribute as Listing 1-12 shows.

Listing 1-12 - Widgets Default Values via HTML Attributes

```
$emailWidget = new sfWidgetFormInput(array(), array('value' => 'Your Email
Here'));

// Generated HTML
<input type="text" name="contact[email]" value="Your Email Here"
id="contact_email" />
```

Listing 1-26

This option works for `input` widgets, but is hard to carry through with `checkbox` or `radio` widgets, and even impossible with a `textarea` widget. The `sfForm` class offers specific methods to define default values for each field in a uniform way for any type of widget.

We recommend to define HTML attributes inside the template and not in the form itself (even if it is possible) to preserve the layers of separation as we will see in Chapter three.

Defining Default Values For Fields

It is often useful to define a default value for each field. For instance, when we display a help message in the field that disappears when the user focuses on the field. Listing 1-13 shows how to define default values via the `setDefault()` and `setDefaults()` methods.

Listing 1-13 - Default Values of the Widgets via the `setDefault()` and `setDefaults()` Methods

```
class ContactForm extends sfForm
{
  public function configure()
  {
    // ...
```

Listing 1-27

```
    $this->setDefault('email', 'Your Email Here');

    $this->setDefaults(array('email' => 'Your Email Here', 'name' => 'Your
Name Here'));
  }
}
```

The `setDefault()` and `setDefaults()` methods are very helpful to define identical default values for every instance of the same form class. If we want to modify an existing object using a form, the default values will depend on the instance, therefore they must be dynamic. Listing 1-14 shows the `sfForm` constructor has a first argument that set default values dynamically.

Listing 1-14 - Default Values of the Widgets via the Constructor of `sfForm`

Listing 1-28
```
public function executeIndex($request)
{
  $this->form = new ContactForm(array('email' => 'Your Email Here', 'name'
=> 'Your Name Here'));

  // ...
}
```

Protection XSS (Cross-Site Scripting)

When setting HTML attributes for widgets, or defining default values, the `sfForm` class automatically protects these values against XSS attacks during the generation of the HTML code. This protection does not depend on the `escaping_strategy` configuration of the `settings.yml` file. If a content has already been protected by another method, the protection will not be applied again.

It also protects the `'` and `"` characters that might invalidate the generated HTML.

Here is an example of this protection:

Listing 1-29
```
$emailWidget = new sfWidgetFormInput(array(), array(
  'value' => 'Hello "World!"',
  'class' => '<script>alert("foo")</script>',
));

// Generated HTML
<input
  value="Hello &quot;World!&quot;"
  class="&lt;script&gt;alert(&quot;foo&quot;)&lt;/script&gt;"
  type="text" name="contact[email]" id="contact_email"
/>
```

Chapter 2

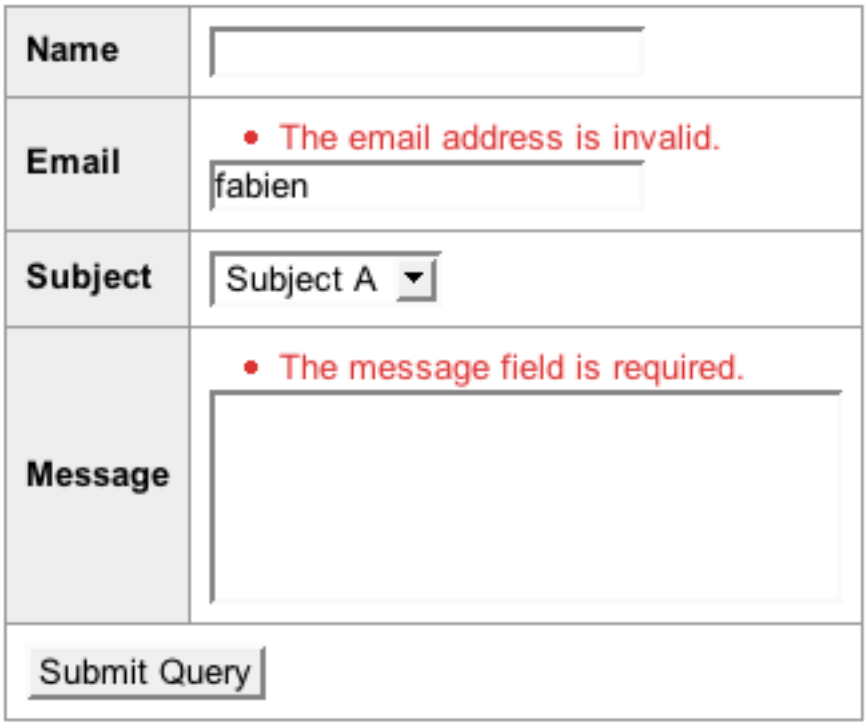
Form Validation

In Chapter 1 we learned how to create and display a basic contact form. In this chapter you will learn how to manage form validation.

Before we start

The contact form created in Chapter 1 is not yet fully functional. What happens if a user submit an invalid email address or if the message the user submits is empty? In these cases, we would like to display error messages to ask the user to correct the input, as shown in Figure 2-1.

Figure 2-1 - Displaying Error Messages



Here are the validation rules to implement for the contact form:

- `name` : optional
- `email` : mandatory, the value must be a valid email address
- `subject`: mandatory, the selected value must be valid to a list of values
- `message`: mandatory, the length of the message must be at least four characters

Why do we need to validate the `subject` field? The `<select>` tag is already binding the user with pre-defined values. An average user can only select one of the displayed choices, but other values can be submitted using tools like the Firefox Developer Toolbar, or by simulating a request with tools like `curl` or `wget`.

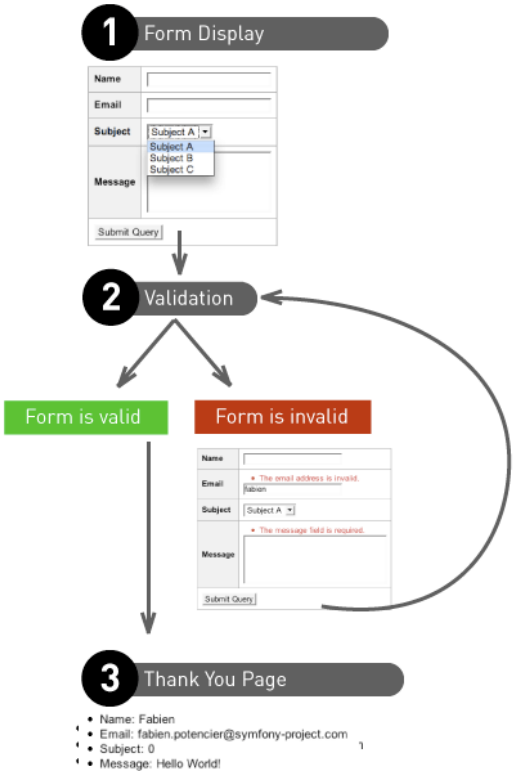
Listing 2-1 shows the template we used in Chapter 1.

Listing 2-1 - The `Contact` Form Template

Listing 2-1
```
// apps/frontend/modules/contact/templates/indexSucces.php
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <?php echo $form ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

Figure 2-2 breaks down the interaction between the application and the user. The first step is to present the form to the user. When the user submits the form, either the input is valid and the user is redirected to the thank you page, or the input includes invalid values and the form is displayed again with error messages.

Figure 2-2 - Interaction between the Application and the User



Validators

A symfony form is made of fields. Each field can be identified by a unique name as we observed in Chapter 1. We connected a widget to each field in order to display it to the user, now let's see how we can apply validation rules to each of the fields.

The `sfValidatorBase` class

The validation of each field is done by objects inheriting from the `sfValidatorBase` class. In order to validate the contact form, we must define validator objects for each of the four fields: `name`, `email`, `subject`, and `message`. Listing 2-2 shows the implementation of these validators in the form class using the `setValidators()` method.

Listing 2-2 - Adding Validators to the `ContactForm` *Class*

Listing 2-2

```
// lib/form/ContactForm.class.php
class ContactForm extends sfForm
{
  protected static $subjects = array('Subject A', 'Subject B', 'Subject
C');

  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
      'subject' => new sfWidgetFormSelect(array('choices' =>
self::$subjects)),
      'message' => new sfWidgetFormTextarea(),
    ));
    $this->widgetSchema->setNameFormat('contact[%s]');

    $this->setValidators(array(
      'name'    => new sfValidatorString(array('required' => false)),
      'email'   => new sfValidatorEmail(),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4)),
    ));
  }
}
```

We use three distinct validators:

- `sfValidatorString`: validates a string
- `sfValidatorEmail` : validates an email
- `sfValidatorChoice`: validates the input value comes from a pre-defined list of choices

Each validator takes a list of options as its first argument. Like the widgets, some of these options are mandatory, some are optional. For instance, the `sfValidatorChoice` validator takes one mandatory option, `choices`. Each validator can also take the options `required` and `trim`, defined by default in the `sfValidatorBase` class:

Option	Default Value	Description
required	`true`	Specifies if the field is mandatory

Option	Default Value	Description
trim	`false`	Automatically removes whitespaces at the beginning and at the end of a string before the validation occurs

Let's see the available options for the validators we have just used:

Validator	Mandatory Options	Optional Options
sfValidatorString		`max_length`
		`min_length`
sfValidatorEmail		`pattern`
sfValidatorChoice	`choices`	

If you try to submit the form with invalid values, you will not see any change in the behavior. We must update the `contact` module to validate the submitted values, as shown in Listing 2-3.

Listing 2-3 - Implementing Validation in the `contact` *Module*

Listing 2-3
```
class contactActions extends sfActions
{
  public function executeIndex($request)
  {
    $this->form = new ContactForm();

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('contact'));
      if ($this->form->isValid())
      {
        $this->redirect('contact/
thankyou?'.http_build_query($this->form->getValues()));
      }
    }
  }

  public function executeThankyou()
  {
  }
}
```

The Listing 2-3 introduces a lot of new concepts:

- In the case of the initial `GET` request, the form is initialized and passed on to the template to display to the user. The form is then in an **initial state**:

Listing 2-4
```
$this->form = new ContactForm();
```

- When the user submits the form with a `POST` request, the `bind()` method binds the form with the user input data and triggers the validation mechanism. The form then changes to a **bound state**.

Listing 2-5
```
if ($request->isMethod('post'))
{
  $this->form->bind($request->getParameter('contact'));
```

- Once the form is bound, it is possible to check its validity using the `isValid()` method:
  - If the return value is `true`, the form is valid and the user can be redirected to the thank you page:

Listing 2-6

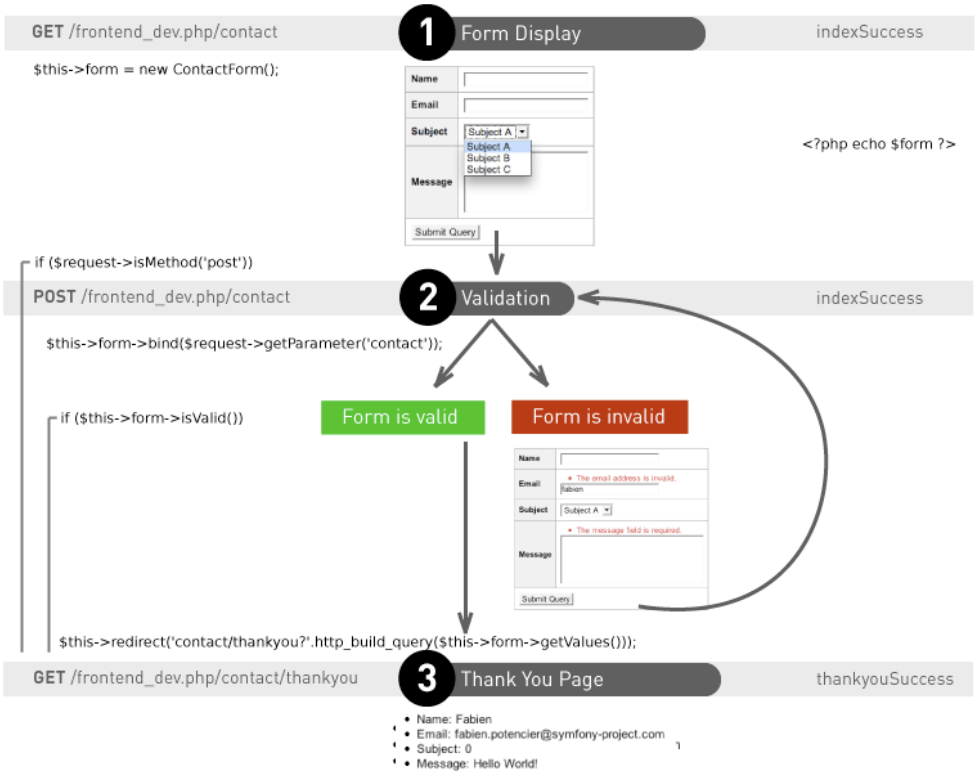
```
if ($this->form->isValid())
{
  $this->redirect('contact/
thankyou?'.http_build_query($this->form->getValues()));
}
```

  - If not, the `indexSuccess` template is displayed as initially. The validation process adds the error messages into the form to be displayed to the user.

When a form is in an initial state, the `isValid()` method always return `false` and the `getValues()` method will always return an empty array.

Figure 2-3 shows the code that is executed during the interaction between the application and the user.

Figure 2-3 - Code executed during the Interaction between the Application and the User



The Purpose of Validators

You might have noticed that during the redirection to the thank you page, we are not using `$request->getParameter('contact')` but `$this->form->getValues()`. In fact, `$request->getParameter('contact')` returns the user data when `$this->form->getValues()` returns the validated data.

If the form is valid, why can not those two statements be identical? Each validator actually has two tasks: a **validation task**, but also a **cleaning task**. The `getValues()` method is in fact returning the validated and cleaned data.

The cleaning process has two main actions: **normalization** and **conversion** of the input data.

We already went over a case of data normalization with the `trim` option. But the normalization action is much more important for a date field for instance. The `sfValidatorDate` validates a date. This validator takes a lot of formats for input (a timestamp, a format based on a regular expression, ...). Instead of simply returning the input value, it converts it by default in the `Y-m-d H:i:s` format. Therefore, the developer is guaranteed to get a stable format, despite the quality of the input format. The system offers a lot of flexibility to the user, and ensures consistency to the developer.

Now, consider a conversion action, like a file upload. A file validation can be done using the `sfValidatorFile`. Once the file is uploaded, instead of returning the name of the file, the validator returns a `sfValidatedFile` object, making it easier to handle the file information. We will see later on in this chapter how to use this validator.

The `getValues()` method returns an array of all the validated and cleaned data. But as retrieving just one value is sometimes helpful, there is also a `getValue()` method: `$email = $this->form->getValue('email')`.

Invalid Form

Whenever there are invalid fields in the form, the `indexSuccess` template is displayed. Figure 2-4 shows what we get when we submit a form with invalid data.

Figure 2-4 - Invalid Form

The call to the `<?php echo $form ?>` statement automatically takes into consideration the error messages associated with the fields, and will automatically populate the users cleaned input data.

When the form is bound to external data by using the `bind()` method, the form switches to a bound state and the following actions are triggered:

- The validation process is executed

- The error messages are stored in the form in order to be available to the template
- The default values of the form are replaced with the users cleaned input data

The information needed to display the error messages or the user input data are easily available by using the `form` variable in the template.

As seen in Chapter 1, we can pass default values to the form class constructor. After the submission of an invalid form, these default values are overridden by the submitted values, so that the user can correct their mistakes. So, never use the input data as default values like in this example: `$this->form->setDefaults($request->getParameter('contact'))`.

Validator Customization

Customizing error messages

As you may have noticed in Figure 2-4, error messages are not really useful. Let's see how to customize them to be more intuitive.

Each validator can add errors to the form. An error consists of an error code and an error message. Every validator has at least the `required` and `invalid` errors defined in the `sfValidatorBase`:

Code	Message	Description
required	`Required.`	The field is mandatory and the value is empty
invalid	`Invalid.`	The field is invalid

Here are the error codes associated to the validators we have already used:

Validator	Error Codes
sfValidatorString	`max_length`
	`min_length`
sfValidatorEmail	
sfValidatorChoice	

Customizing error messages can be done by passing a second argument when creating the validation objects. Listing 2-4 customizes several error messages and Figure 2-5 shows customized error messages in action.

Listing 2-4 - Customizing Error Messages

Listing 2-7

```
class ContactForm extends sfForm
{
  protected static $subjects = array('Subject A', 'Subject B', 'Subject
C');

  public function configure()
  {
    // ...

    $this->setValidators(array(
      'name'    => new sfValidatorString(array('required' => false)),
      'email'   => new sfValidatorEmail(array(), array('invalid' => 'The
```

```
email address is invalid.')),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4),
array('required' => 'The message field is required.')),
    ));
  }
}
```

Figure 2-5 - Customized Error Messages

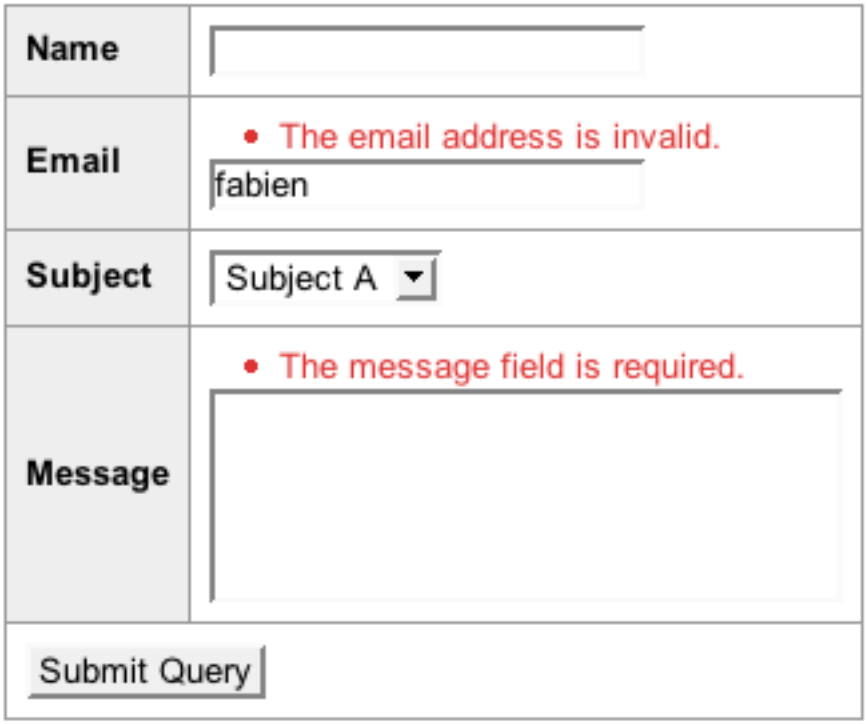
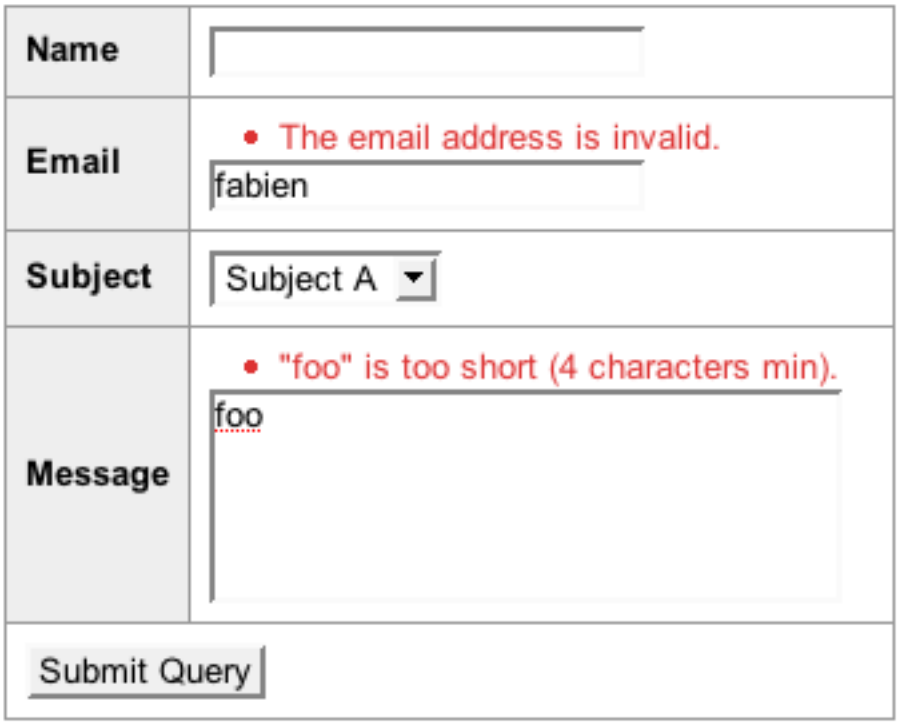


Figure 2-6 shows the error message you get if you try to submit a message too short (we set the minimum length to 4 characters).

Figure 2-6 - Too short Message Error



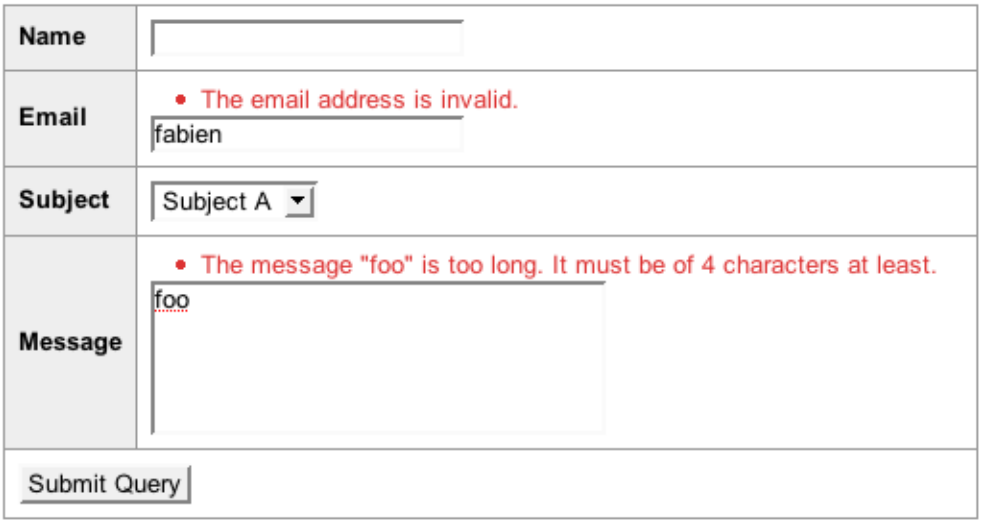
The default error message related to this error code (`min_length`) is different from the messages we already went over, since it implements two dynamic values: the user input data (`foo`) and the minimum number of characters allowed for this field (`4`). Listing 2-5 customizes this message using theses dynamic values and Figure 2-7 shows the result.

Listing 2-5 - Customizing the Error Messages with Dynamic Values

Listing 2-8

```
class ContactForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->setValidators(array(
      'name'    => new sfValidatorString(array('required' => false)),
      'email'   => new sfValidatorEmail(array(), array('invalid' => 'Email
address is invalid.')),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4), array(
        'required'   => 'The message field is required',
        'min_length' => 'The message "%value%" is too short. It must be of
%min_length% characters at least.',
      )),
    ));
  }
}
```

Figure 2-7 - Customized Error Messages with Dynamic Values



Each error message can use dynamic values, enclosing the value name with the percent character (`%`). Available values are usually the user input data (`value`) and the option values of the validator related to the error.

If you want to review all the error codes, options, and default message of a validator, please refer to the API online documentation (http://www.symfony-project.org/api/1_1/[4]). Each code, option and error message are detailed there, along with the default values (for instance, the `sfValidatorString` validator API is available at http://www.symfony-project.org/api/1_1/sfValidatorString[5]).

Validators Security

By default, a form is valid only if every field submitted by the user has a validator. This ensures that each field has its validation rules and that it is not possible to inject values for fields that are not defined in the form.

To help understand this security rule, let's consider a user object as shown in Listing 2-6.

Listing 2-6 - The `User` *Class*

Listing 2-9

```
class User
{
  protected
    $name = '',
    $is_admin = false;

  public function setFields($fields)
  {
    if (isset($fields['name']))
    {
      $this->name = $fields['name'];
    }

    if (isset($fields['is_admin']))
    {
```

4. http://www.symfony-project.org/api/1_1/
5. http://www.symfony-project.org/api/1_1/sfValidatorString

```
      $this->is_admin = $fields['is_admin'];
    }
  }

  // ...
}
```

A `User` object is composed of two properties, the user name (`name`), and a boolean that stores the administrator status (`is_admin`). The `setFields()` method updates both properties. Listing 2-7 shows the form related to the `User` class, allowing the user to modify the `name` property only.

Listing 2-7 - `User` *Form*

```
class UserForm extends sfForm                                    Listing 2-10
{
  public function configure()
  {
    $this->setWidgets(array('name' => new sfWidgetFormInputString()));
    $this->widgetSchema->setNameFormat('user[%s]');

    $this->setValidators(array('name' => new sfValidatorString()));
  }
}
```

Listing 2-8 shows an implementation of the `user` module using the previously defined form allowing the user to modify the name field.

Listing 2-8 - `user` *Module Implementation*

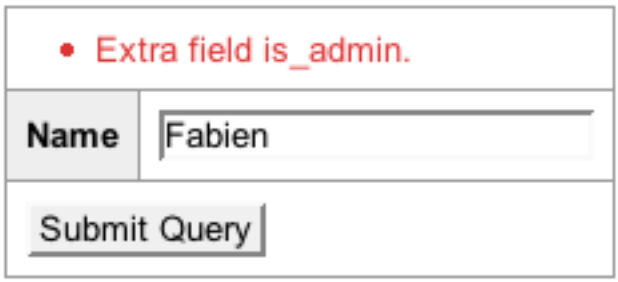
```
class userActions extends sfActions                              Listing 2-11
{
  public function executeIndex($request)
  {
    $this->form = new UserForm();

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('user'));
      if ($this->form->isValid())
      {
        $user = // retrieving the current user

        $user->setFields($this->form->getValues());

        $this->redirect('...');
      }
    }
  }
}
```

Without any protection, if the user submits a form with a value for the `name` field, and also for the `is_admin` field, then our code is vulnerable. This is easily accomplished using a tool like Firebug. In fact, the `is_admin` value is always valid, because the field does not have any validator associated with it in the form. Whatever the value is, the `setFields()` method will update not only the `name` property, but also the `is_admin` property.

If you test out this code passing a value for both the `name` and `is_admin` fields, you'll get an "Extra field name." global error, as shown in Figure 2-8. The system generated an error

because some submitted fields does not have any validator associated with themselves; the `is_admin` field is not defined in the `UserForm` form.

Figure 2-8 - Missing Validator Error



All the validators we have seen so far generate errors associated with fields. Where can this global error come from? When we use the `setValidators()` method, symfony creates a `sfValidatorSchema` object. The `sfValidatorSchema` defines a collection of validators. The call to `setValidators()` is equivalent to the following code:

Listing 2-12
```
$this->setValidatorSchema(new sfValidatorSchema(array(
  'email'   => new sfValidatorEmail(),
  'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
  'message' => new sfValidatorString(array('min_length' => 4)),
)));
```

The `sfValidatorSchema` has two validation rules enabled by default to protect the collection of validators. These rules can be configured with the `allow_extra_fields` and `filter_extra_fields` options.

The `allow_extra_fields` option, which is set to `false` by default, checks that every user input data has a validator. If not, an "Extra field name." global error is thrown, as shown in the previous example. When developing, this allows developers to be warned if one forgets to explicitly validate a field.

Let's get back to the contact form. Let's change the validation rules by changing the `name` field into a mandatory field. Since the default value of the `required` option is `true`, we could change the `name` validator to:

Listing 2-13
```
$nameValidator = new sfValidatorString();
```

This validator has no impact as it has neither a `min_length` nor a `max_length` option. In this case, we could also replace it with an empty validator:

Listing 2-14
```
$nameValidator = new sfValidatorPass();
```

Instead of defining an empty validator, we could get rid of it, but the protection by default we previously went over prevents us from doing so. Listing 2-9 shows how to disable the protection using the `allow_extra_fields` option.

Listing 2-9 - Disable the `allow_extra_fields` *Protection*

Listing 2-15
```
class ContactForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->setValidators(array(
```

```
      'email'   => new sfValidatorEmail(),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4)),
    ));

    $this->validatorSchema->setOption('allow_extra_fields', true);
  }
}
```

You should now be able to validate the form as shown in Figure 2-9.

Figure 2-9 - Validating with `allow_extra_fields` *set to* `true`

- Name:
- Email: fabien.potencier@symfony-project.com
- Subject: 0
- Message: Hello World!

If you have a closer look, you will notice that even if the form is valid, the value of the `name` field is empty in the thank you page, despite any value that was submitted. In fact, the value wasn't even set in the array sent back by `$this->form->getValues()`. Disabling the `allow_extra_fields` option let us get rid of the error due to the lack of validator, but the `filter_extra_fields` option, which is set to `true` by default, filters those values, removing them from the validated values. It is of course possible to change this behavior, as shown in Listing 2-10.

Listing 2-10 - Disabling the `filter_extra_fields` *protection*

Listing 2-16

```
class ContactForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->setValidators(array(
      'email'   => new sfValidatorEmail(),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4)),
    ));

    $this->validatorSchema->setOption('allow_extra_fields', true);
    $this->validatorSchema->setOption('filter_extra_fields', false);
  }
}
```

You should now be able to validate your form and retrieve the input value in the thank you page.

We will see in Chapter 4 that these protections can be used to safely serialize Propel objects from form values.

Logical Validators

Several validators can be defined for a single field by using logical validators:

- `sfValidatorAnd`: To be valid, the field must pass all validators
- `sfValidatorOr` : To be valid, the filed must pass at least one validator

The constructors of the logical operators take a list of validators as their first argument. Listing 2-11 uses the `sfValidatorAnd` to associate two required validators to the `name` field.

Listing 2-11 - Using the `sfValidatorAnd` *validator*

Listing 2-17
```
class ContactForm extends sfForm
{
 public function configure()
 {
    // ...

    $this->setValidators(array(
      // ...
      'name' => new sfValidatorAnd(array(
        new sfValidatorString(array('min_length' => 5)),
        new sfValidatorRegex(array('pattern' => '/[\w- ]+/')),
      )),
    ));
  }
}
```

When submitting the form, the `name` field input data must be made of at least five characters **and** match the regular expression (`[\w- ]+`).

As logical validators are validators themselves, they can be combined to define advanced logical expressions as shown in Listing 2-12.

Listing 2-12 - Combining several logical Operators

Listing 2-18
```
class ContactForm extends sfForm
{
 public function configure()
 {
    // ...

    $this->setValidators(array(
      // ...
      'name' => new sfValidatorOr(array(
        new sfValidatorAnd(array(
          new sfValidatorString(array('min_length' => 5)),
          new sfValidatorRegex(array('pattern' => '/[\w- ]+/')),
        )),
        new sfValidatorEmail(),
      )),
    ));
  }
}
```

Global Validators

Each validator we went over so far are associated with a specific field and lets us validate only one value at a time. By default, they behave disregarding other data submitted by the user, but sometimes the validation of a field depends on the context or depends on many other field values. For example, a global validator is needed when two passwords must be the same, or when a start date must be before an end date.

In both of these cases, we must use a global validator to validate the input user data in their context. We can store a global validator before or after the individual field validation by using a pre-validator or a post-validator respectively. It is usually better to use a post-validator, because the data is already validated and cleaned, i.e. in a normalized format. Listing 2-13 shows how to implement the two passwords comparison using the `sfValidatorSchemaCompare` validator.

Listing 2-13 - Using the `sfValidatorSchemaCompare` *Validator*

Listing 2-19
```
$this->validatorSchema->setPostValidator(new
sfValidatorSchemaCompare('password', sfValidatorSchemaCompare::EQUAL,
'password_again'));
```

As of symfony 1.2, you can also use the "natural" PHP operators instead of the `sfValidatorSchemaCompare` class constants. The previous example is equivalent to:

Listing 2-20
```
$this->validatorSchema->setPostValidator(new
sfValidatorSchemaCompare('password', '==', 'password_again'));
```

The `sfValidatorSchemaCompare` class inherits from the `sfValidatorSchema` validator, like every global validator. `sfValidatorSchema` is itself a global validator since it validates the whole user input data, passing to other validators the validation of each field.

Listing 2-14 shows how to use a single validator to validate that a start date is before an end date, customizing the error message.

Listing 2-14 - Using the `sfValidatorSchemaCompare` *Validator*

Listing 2-21
```
$this->validatorSchema->setPostValidator(
  new sfValidatorSchemaCompare('start_date',
sfValidatorSchemaCompare::LESS_THAN_EQUAL, 'end_date',
    array(),
    array('invalid' => 'The start date ("%left_field%") must be before the
end date ("%right_field%")')
  )
);
```

Using a post-validator ensures that the comparison of the two dates will be accurate. Whatever date format was used for the input, the validation of the `start_date` and `end_date` fields will always be converted to values in a comparable format (`Y-m-d H:i:s` by default).

By default, pre-validators and post-validators return global errors to the form. Nevertheless, some of them can associate an error to a specific field. For instance, the `throw_global_error` option of the `sfValidatorSchemaCompare` validator can choose between a global error (Figure 2-10) or an error associated to the first field (Figure 2-11). Listing 2-15 shows how to use the `throw_global_error` option.

Listing 2-15 - Using the `throw_global_error` *Option*

Listing 2-22
```
$this->validatorSchema->setPostValidator(
  new sfValidatorSchemaCompare('start_date',
sfValidatorSchemaCompare::LESS_THAN_EQUAL, 'end_date',
    array('throw_global_error' => true),
    array('invalid' => 'The start date ("%left_field%") must be before the
end date ("%right_field%")')
  )
);
```

Figure 2-10 - Global Error for a Global Validator

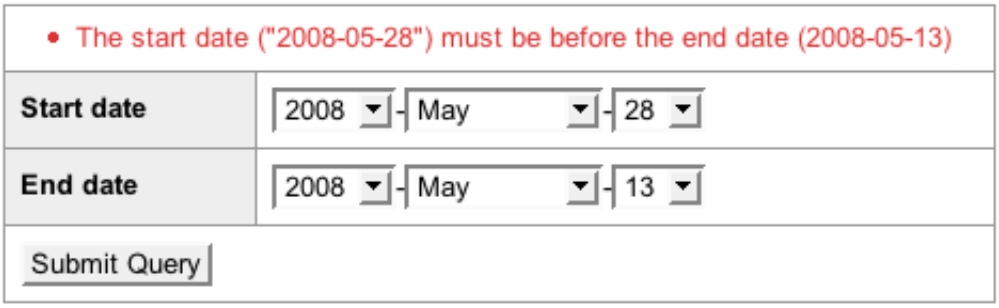
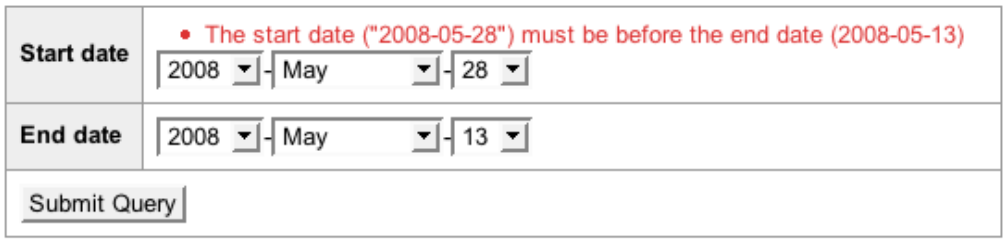


Figure 2-11 - Local Error for a Global Validator



At last, using a logical validator allows you to combine several post-validators as shown Listing 2-16.

Listing 2-16 - Combining several Post-Validators with a logical Validator

Listing 2-23
```
$this->validatorSchema->setPostValidator(new sfValidatorAnd(array(
  new sfValidatorSchemaCompare('start_date',
sfValidatorSchemaCompare::LESS_THAN_EQUAL, 'end_date'),
  new sfValidatorSchemaCompare('password',
sfValidatorSchemaCompare::EQUAL, 'password_again'),
)));
```

File Upload

Dealing with file upload in PHP, like in every web oriented language, involves handling both HTML code and server-side file retrieving. In this section we will see the tools the form framework has to offer to the developer to make their life easier. We will also see how to not fall into common traps.

Let's change the contact form to allow attaching a file to be attached to the message. To do this, we will add a `file` field as shown in Listing 2-17.

Listing 2-17 - Adding a `file` Field to the `ContactForm` form

Listing 2-24
```
// lib/form/ContactForm.class.php
class ContactForm extends sfForm
{
  protected static $subjects = array('Subject A', 'Subject B', 'Subject
C');

  public function configure()
  {
    $this->setWidgets(array(
      'name'    => new sfWidgetFormInput(),
      'email'   => new sfWidgetFormInput(),
```

```
      'subject' => new sfWidgetFormSelect(array('choices' =>
self::$subjects)),
      'message' => new sfWidgetFormTextarea(),
      'file'    => new sfWidgetFormInputFile(),
    ));
    $this->widgetSchema->setNameFormat('contact[%s]');

    $this->setValidators(array(
      'name'    => new sfValidatorString(array('required' => false)),
      'email'   => new sfValidatorEmail(),
      'subject' => new sfValidatorChoice(array('choices' =>
array_keys(self::$subjects))),
      'message' => new sfValidatorString(array('min_length' => 4)),
      'file'    => new sfValidatorFile(),
    ));
  }
}
```

When there is a `sfWidgetFormInputFile` widget in a form allowing to upload a file, we must also add an `enctype` attribute to the `form` tag as shown in Listing 2-18.

Listing 2-18 - Modifying the Template to take the `file` Field into account

Listing 2-25

```
<form action="<?php echo url_for('contact/index') ?>" method="POST"
enctype="multipart/form-data">
  <table>
    <?php echo $form ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

If you dynamically generate the template associated to a form, the `isMultipart()` method of the form object return `true`, if it needs the `enctype` attribute.

Information about uploaded files are not stored with the other submitted values in PHP. It is then necessary to modify the call to the `bind()` method to pass on this information as a second argument, as shown in Listing 2-19.

Listing 2-19 - Passing uploaded Files to the `bind()` Method

Listing 2-26

```
class contactActions extends sfActions
{
  public function executeIndex($request)
  {
    $this->form = new ContactForm();

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('contact'),
$request->getFiles('contact'));
      if ($this->form->isValid())
      {
        $values = $this->form->getValues();
        // do something with the values
```

```
        // ...
      }
    }
  }

  public function executeThankyou()
  {
  }
}
```

Now that the form is fully operational, we still need to change the action in order to store the uploaded file on disk. As we observed at the beginning of this chapter, the `sfValidatorFile` converts the information related to the uploaded file to a `sfValidatedFile` object. Listing 2-20 shows how to handle this object to store the file in the `web/uploads` directory.

Listing 2-20 - Using the `sfValidatedFile` *Object*

Listing 2-27

```
if ($this->form->isValid())
{
  $file = $this->form->getValue('file');

  $filename = 'uploaded_'.sha1($file->getOriginalName());
  $extension = $file->getExtension($file->getOriginalExtension());
  $file->save(sfConfig::get('sf_upload_dir').'/'.$filename.$extension);

  // ...
}
```

The following table lists all the `sfValidatedFile` object methods:

Method	Description
save()	Saves the uploaded file
isSaved()	Returns `true` if the file has been saved
getSavedName()	Returns the name of the saved file
getExtension()	Returns the extension of the file, according to the mime type
getOriginalName()	Returns the name of the uploaded file
getOriginalExtension()	Returns the extension of the uploaded file name
getTempName()	Returns the path of the temporary file
getType()	Returns the mime type of the file
getSize()	Returns the size of the file

The mime type provided by the browser during the file upload is not reliable. In order to ensure maximum security, the functions `finfo_open` and `mime_content_type`, and the `file` tool are used in turn during the file validation. As a last resort, if any of the functions can not guess the mime type, or if the system does not provide them, the browser mime type is taken into account. To add or change the functions that guess the mime type, just pass the `mime_type_guessers` option to the `sfValidatorFile` constructor.

Chapter 3

Forms for Web Designers

We observed in Chapter 1 and Chapter 2 how to create forms using widgets and validation rules. We used the `<?php echo $form ?>` statement to display them. This statement allows developers to code the application logic without thinking about how it will look in the end. Changing the template every time you modify a field (name, widget...) or even add one is not necessary. This statement is well suited for prototyping and the initial development phase, when the developer has to focus on the model and the business logic.

Once the object model is stabilized and the style guidelines are in place, the web designer can go back and format the various application forms.

Before starting this chapter, you should be well acquainted with symfony's templating system and view layer. To do so, you can read the Inside the View Layer[6] chapter of the "The Definitive Guide to symfony" book.

Symfony's form system is built according to the MVC model. The MVC pattern helps decouple every task of a development team: The developers create the forms and handle their life cycles, and the Web designers format and style them. The separation of concerns will never be a replacement for the communication within the project team.

Before we start

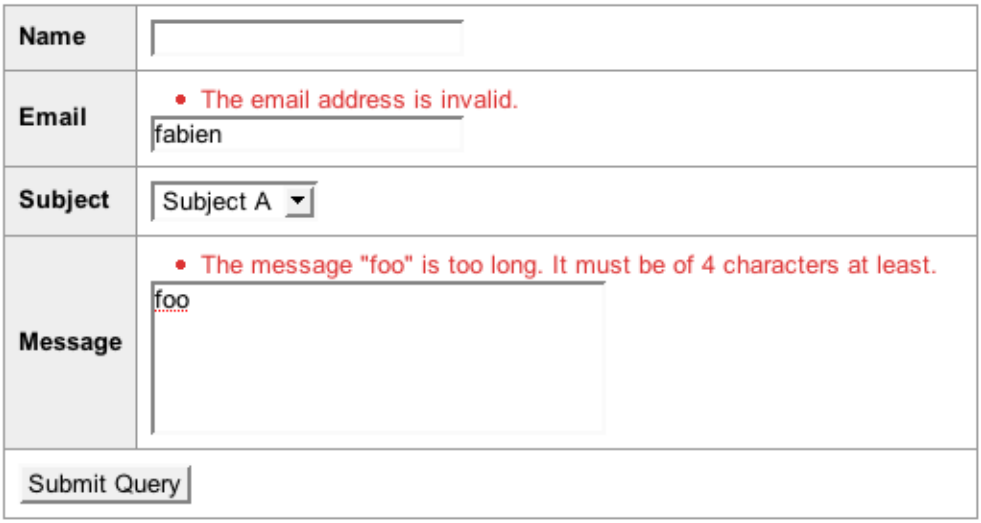
We will now go over the contact form elaborated in Chapters 1 and 2 (Figure 3-1). Here is a technical overview for Web Designers who will only read this chapter:

- The form is made of four fields: `name`, `email`, `subject`, and `message`.
- The form is handled by the `contact` module.
- The `index` action passes on to the template a `form` variable representing the form.

This chapter aims to show the available possibilities to customize the prototype template we used to display the form (Listing 3-1).

Figure 3-1 - The Contact Form

6. `http://www.symfony-project.org/book/1_1/07-Inside-the-View-Layer`



Listing 3-1 - The Prototype Template diplaying the Contact Form

Listing 3-1
```
// apps/frontend/modules/contact/templates/indexSuccess.php
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <?php echo $form ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

File Upload

Whenever you use a field to upload a file in a form, you must add an `enctype` attribute to the `form` tag:

Listing 3-2
```
<Form action="<?php echo url_for('contact/index') ?>" method="POST"
enctype="multipart/data">
```

The `isMultipart()` method of the `form` object returns `true` if the form needs this attribute:

Listing 3-3
```
<Form action="<?php echo url_for('contact/index') ?>" method="POST" <?php
$form->isMultipart() and print 'enctype="multipart/form-data"' ?>>
```

The Prototype Template

As of now, we have used the `<?php echo $form ?>` statement in the prototype template in order to automatically generate the HTML needed to display the form.

A form is made of fields. At the template level, each field is made of three elements:

- The label
- The form tag

- The potential error messages

The `<?php echo $form ?>` statement automatically generates all these elements, as Listing 3-2 shows in the case of an invalid submission.

Listing 3-2 - Generated Template in case of invalid Submission

Listing 3-4

```
<form action="/frontend_dev.php/contact" method="POST">
  <table>
    <tr>
      <th><label for="contact_name">Name</label></th>
      <td><input type="text" name="contact[name]" id="contact_name" /></td>
    </tr>
    <tr>
      <th><label for="contact_email">Email</label></th>
      <td>
        <ul class="error_list">
          <li>This email address is invalid.</li>
        </ul>
        <input type="text" name="contact[email]" value="fabien"
id="contact_email" />
      </td>
    </tr>
    <tr>
      <th><label for="contact_subject">Subject</label></th>
      <td>
        <select name="contact[subject]" id="contact_subject">
          <option value="0" selected="selected">Subject A</option>
          <option value="1">Subject B</option>
          <option value="2">Subject C</option>
        </select>
      </td>
    </tr>
    <tr>
      <th><label for="contact_message">Message</label></th>
      <td>
        <ul class="error_list">
          <li>The message "foo" is too short. It must be of 4 characters
at least.</li>
        </ul>
        <textarea rows="4" cols="30" name="contact[message]"
id="contact_message">foo</textarea>
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

There is an additional shortcut to generate the opening form tag for the form: `echo $form->renderFormTag(url_for('contact/index'))`. It also allows passing any number of additional attributes to the form tag more easily by proving an array. The downside of using this shortcut is that design tools will have more troubles detecting the form properly.

Let's break this code down. Figure 3-2 underlines the `<tr>` rows produced for each field.

Figure 3-2 - The Form Split by Field

Three pieces of HTML code have been generated for each field (Figure 3-3), matching the three elements of the field. Here is the HTML code generated for the `email` field:

- The **label**

Listing 3-5
```
<label for="contact_email">Email</label>
```

- The **form tag**

Listing 3-6
```
<input type="text" name="contact[email]" value="fabien"
id="contact_email" />
```

- The **error messages**

Listing 3-7
```
<ul class="error_list">
  <li>The email address is invalid.</li>
</ul>
```

Figure 3-3 - Decomposition of the `email` *Field*

Every field has a generated `id` attribute which allows developers to add styles or JavaScript behaviors very easily.

The Prototype Template Customization

The `<?php echo $form ?>` statement can be enough for simple forms like the contact form. And, as a matter of fact, it is just a shortcut for the `<?php echo $form->render() ?>` statement.

The usage of the `render()` method allows to pass on HTML attributes as an argument for each field. Listing 3-3 shows how to add a class to the `email` field.

Listing 3-3 - Customization of the HTML Attributes using the `render()` *Method.*

Listing 3-8

```
<?php echo $form->render(array('email' => array('class' => 'email'))) ?>

// Generated HTML
<input type="text" name="contact[email]" value="" id="contact_email"
class="email" />
```

This allows to customize the form styles but does not provide the level of flexibility needed to customize the organization of the fields in the page.

The Display Customization

Beyond the global customization allowed by the `render()` method, let's see now how to break the display of each field down to gain in flexibility.

Using the `renderRow()` method on a field

The first way to do it is to generate every field individually. In fact, the `<?php echo $form ?>` statement is equivalent to calling the `renderRow()` method four times on the form, as shown in Listing 3-4.

Listing 3-4 - `renderRow()` *Usage*

Listing 3-9

```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <?php echo $form['name']->renderRow() ?>
    <?php echo $form['email']->renderRow() ?>
    <?php echo $form['subject']->renderRow() ?>
    <?php echo $form['message']->renderRow() ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

We access each field using the `form` object as a PHP array. The `email` field can then be accessed via `$form['email']`. The `renderRow()` method displays the field as an HTML table row. The expression `$form['email']->renderRow()` generates a row for the `email` field. By

repeating the same kind of code for the three other fields `subject`, `email`, and `message`, we complete the display of the form.

How can an Object behave like an Array?

Since PHP version 5, objects can be given the same behavior than an PHP array. The `sfForm` class implements the ArrayAccess behavior to grant access to each field using a simple and short syntax. The key of the array is the field name and the returned value is the associated widget object:

Listing 3-10
```
<?php echo $form['email'] ?>

// Syntax that should have been used if sfForm didn't implement the
ArrayAccess interface.
<?php echo $form->getField('email') ?>
```

However, as every variable must be read-only in templates, any attempt to modify the field will throw a `LogicException` exception:

Listing 3-11
```
<?php $form['email'] = ... ?>
<?php unset($form['email']) ?>
```

This current template and the original template we started with are functionally identical. However, if the display is the same, the customization is now easier. The `renderRow()` method takes two arguments: an HTML attributes array and a label name. Listing 3-5 uses those two arguments to customize the form (Figure 3-4 shows the rendering).

Listing 3-5 - Using the `renderRow()` Method's Arguments to customize the display.

Listing 3-12
```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <?php echo $form['name']->renderRow() ?>
    <?php echo $form['email']->renderRow(array('class' => 'email')) ?>
    <?php echo $form['subject']->renderRow() ?>
    <?php echo $form['message']->renderRow(array(), 'Your message') ?>
    <tr>
      <td colspan="2">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

Figure 3-4 - Customization of the Form display using the `renderRow()` Method

Name	
Email	• The email address is invalid. fabien
Subject	Subject A
Message	• The message "foo" is too long. It must be of 4 characters at least. foo
Submit Query	

Let's have a closer look at the arguments sent to `renderRow()` in order to generate the `email` field:

- `array('class' => 'email')` adds the `email` class to the `<input>` tag

It works the same way with the `message` field:

- `array()` mean that we does not want to add any HTML attributes to the `<textarea>` tag
- `'Your message'` replaces the default label name

Every `renderRow()` method argument is optional, so none of them are required as we did for the `name` and `subject` fields.

Even if the `renderRow()` method helps customizing the elements of each field, the rendering is limited by the HTML code decorating these elements as shown in Figure 3-5.

Figure 3-5 - HTML Structure used by `renderRow()` *and* `render()`

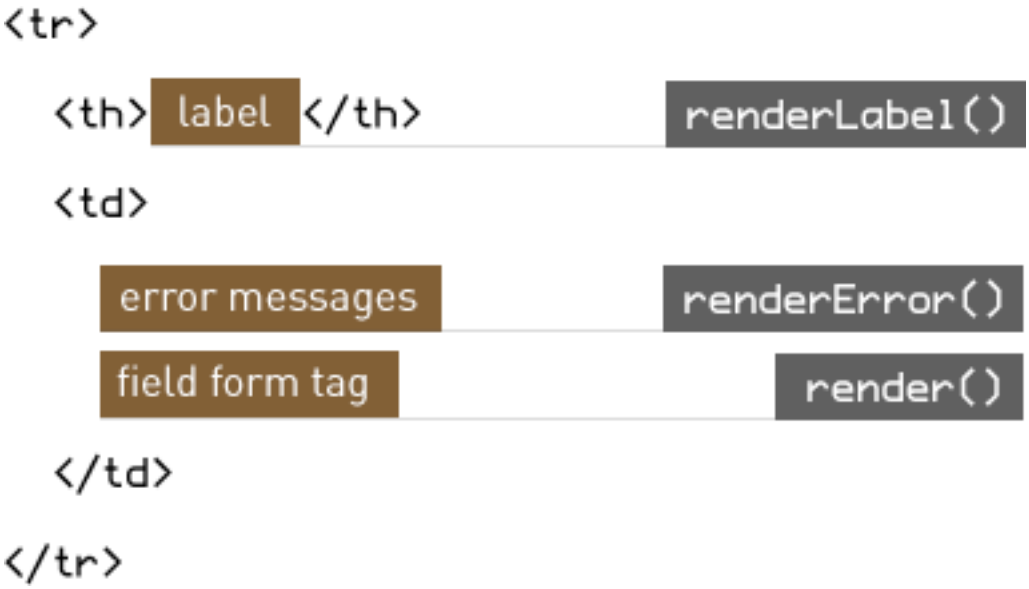
How to change the Structure Format used by the Prototyping?

By default, symfony uses an HTML array to display a form. This behavior can be changed using specific *formatters*, whether they're built-in or specifically developed to suit the project. To create a formatter, you need to create a class as described in Chapter 5.

In order to break free from this structure, each field has methods generating its elements, as shown in Figure 3-6:

- `renderLabel()` : the label (the`<label>` tag tied to the field)
- `render()` : the field tag itself (the `<input>` tag for instance)
- `renderError()` : error messages (as a `<ul class="error_list">` list)

Figure 3-6 - Methods available to customize a Field



These methods will be explained at the end of this chapter.

Using the `render()` method on a field

Suppose we want to display the form with two columns. As shown in Figure 3-7, the `name` and `email` fields stand on the same row, when the `subject` and `message` fields stand on their own row.

Figure 3-7 - Displaying the Form with several Rows

We have to be able to generate each element of a field separately to do so. We already observed that we could use the `form` object as an associative array to access a field, using the field name as key. For example, the `email` field can be accessed with `$form['email']`. Listing 3-6 shows how to implement the form with two rows.

Listing 3-6 - Customizing the Display with two Columns

Listing 3-13

```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <tr>
      <th>Name:</th>
      <td><?php echo $form['name']->render() ?></td>
      <th>Email:</th>
      <td><?php echo $form['email']->render() ?></td>
    </tr>
    <tr>
      <th>Subject:</th>
      <td colspan="3"><?php echo $form['subject']->render() ?></td>
    </tr>
    <tr>
      <th>Message:</th>
      <td colspan="3"><?php echo $form['message']->render() ?></td>
    </tr>
    <tr>
      <td colspan="4">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

Just like the explicit use of the `render()` method on a field is not mandatory when using `<?php echo $form ?>`, we can rewrite the template as in Listing 3-7.

Listing 3-7 - Simplifying the two Columns customization

Listing 3-14

```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <tr>
      <th>Name:</th>
      <td><?php echo $form['name'] ?></td>
      <th>Email:</th>
      <td><?php echo $form['email'] ?></td>
    </tr>
    <tr>
      <th>Subject:</th>
      <td colspan="3"><?php echo $form['subject'] ?></td>
    </tr>
    <tr>
      <th>Message:</th>
      <td colspan="3"><?php echo $form['message'] ?></td>
    </tr>
    <tr>
      <td colspan="4">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

Like with the form, each field can be customized by passing an HTML attribute array to the `render()` method. Listing 3-8 shows how to modify the HTML class of the `email` field.

Listing 3-8 - Modifying the HTML Attributes using the `render()` Method

Listing 3-15
```
<?php echo $form['email']->render(array('class' => 'email')) ?>

// Generated HTML
<input type="text" name="contact[email]" class="email" id="contact_email"
/>
```

Using the `renderLabel()` method on a field

We did not generate labels during the customization in the previous paragraph. Listing 3-9 uses the `renderLabel()` method in order to generate a label for each field.

Listing 3-9 - Using `renderLabel()`

Listing 3-16
```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <tr>
      <th><?php echo $form['name']->renderLabel() ?>:</th>
      <td><?php echo $form['name'] ?></td>
      <th><?php echo $form['email']->renderLabel() ?>:</th>
      <td><?php echo $form['email'] ?></td>
    </tr>
    <tr>
      <th><?php echo $form['subject']->renderLabel() ?>:</th>
      <td colspan="3"><?php echo $form['subject'] ?></td>
    </tr>
    <tr>
      <th><?php echo $form['message']->renderLabel() ?>:</th>
      <td colspan="3"><?php echo $form['message'] ?></td>
    </tr>
    <tr>
      <td colspan="4">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

The label name is automatically generated from the field name. It can be customized by passing an argument to the `renderLabel()` method as shown in Listing 3-10.

Listing 3-10 - Modifying the Label Name

Listing 3-17
```
<?php echo $form['message']->renderLabel('Your message') ?>

// Generated HTML
<label for="contact_message">Your message</label>
```

What's the point of the `renderLabel()` method if we send the label name as an argument? Why does not we simply use an HTML `label` tag? That is because the `renderLabel()` method generates the `label` tag and automatically adds a `for` attribute set to the identifier of the linked field (`id`). This ensures that the field will be accessible; when clicking on the label, the field is automatically focused:

Listing 3-18

```
<label for="contact_email">Email</label>
<input type="text" name="contact[email]" id="contact_email" />
```

Moreover, HTML attributes can be added by passing a second argument to the `renderLabel()` method:

Listing 3-19

```
<?php echo $form['send_notification']->renderLabel(null, array('class' =>
'inline')) ?>

// Generated HTML
<label for="contact_send_notification" class="inline">Send
notification</label>
```

In this example, the first argument is `null` so that the automatic generation of the label text is preserved.

Using the `renderError()` method on a field

The current template does not handle error messages. Listing 3-11 restores them using the `renderError()` method.

Listing 3-11 - Displaying Error Messages using the `renderError()` Method

Listing 3-20

```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <tr>
      <th><?php echo $form['name']->renderLabel() ?>:</th>
      <td>
        <?php echo $form['name']->renderError() ?>
        <?php echo $form['name'] ?>
      </td>
      <th><?php echo $form['email']->renderLabel() ?>:</th>
      <td>
        <?php echo $form['email']->renderError() ?>
        <?php echo $form['email'] ?>
      </td>
    </tr>
    <tr>
      <th><?php echo $form['subject']->renderLabel() ?>:</th>
      <td colspan="3">
        <?php echo $form['subject']->renderError() ?>
        <?php echo $form['subject'] ?>
      </td>
    </tr>
    <tr>
      <th><?php echo $form['message']->renderLabel() ?>:</th>
      <td colspan="3">
        <?php echo $form['message']->renderError() ?>
        <?php echo $form['message'] ?>
      </td>
    </tr>
    <tr>
      <td colspan="4">
        <input type="submit" />
      </td>
    </tr>
  </table>
</form>
```

Fine-grained customization of error messages

The `renderError()` method generates the list of the errors associated with a field. It generates HTML code only if the field has some error. By default, the list is generated as an unordered HTML list (`<ul>`).

Even if this behavior suits most of the common cases, the `hasError()` and `getError()` methods allow us to access the errors directly. Listing 3-12 shows how to customize the error messages for the `email` field.

Listing 3-12 - Accessing Error Messages

Listing 3-21
```
<?php if ($form['email']->hasError()): ?>
  <ul class="error_list">
    <?php foreach ($form['email']->getError() as $error): ?>
      <li><?php echo $error ?></li>
    <?php endforeach; ?>
  </ul>
<?php endif; ?>
```

In this example, the generated code is exactly the same as the code generated by the `renderError()` method.

Handling hidden fields

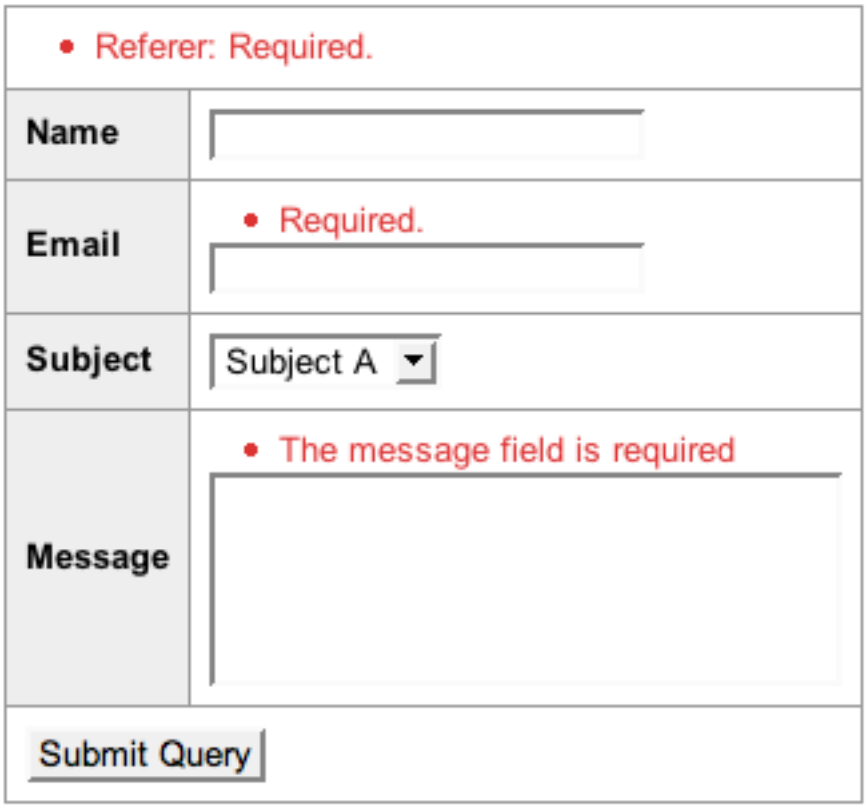
Suppose now there is a mandatory hidden field `referrer` in the form. This field stores the referrer page of the user when accessing the form. The `<?php echo $form ?>` statement generates the HTML code for hidden fields and adds it when generating the last visible field, as shown in Listing 3-13.

Listing 3-13 - Generating the Hidden Fields Code

Listing 3-22
```
<tr>
  <th><label for="contact_message">Message</label></th>
  <td>
    <textarea rows="4" cols="30" name="contact[message]"
id="contact_message"></textarea>
    <input type="hidden" name="contact[referrer]" id="contact_referrer" />
  </td>
</tr>
```

As you can notice in the generated code for the `referrer` hidden field, only the tag element has been added to the output. It makes sense not to generate a label. What about the potential errors that could occur with this field? Even if the field is hidden, it can be corrupted during the processing either on purpose, or because there is an error in the code. These errors are not directly connected to the `referrer` field, but are summed up with the global errors. We will see in Chapter 5 that the notion of global errors is extended also to other cases. Figure 3-8 shows how the error message is displayed when an error occurs on the `referrer` field, and Listing 3-14 shows the code generated for those errors.

Figure 3-8 - Displaying the Global Error Messages



Listing 3-14 - Generating Global Error Messages

Listing 3-23

```
<tr>
  <td colspan="2">
    <ul class="error_list">
      <li>Referrer: Required.</li>
    </ul>
  </td>
</tr>
```

Whenever you customize a form, do not forget to implement hidden fields and global error messages.

Handling global errors

There are three kinds of error for a form:

- Errors associated to a specific field
- Global errors
- Errors from hidden fields or fields that are not actually displayed in the form. Those are summed up with the global errors.

We already went over the implementation of error messages associated with a field, and Listing 3-15 shows the implementation of global error messages.

Listing 3-15 - Implementing global error messages

Listing 3-24

```
<form action="<?php echo url_for('contact/index') ?>" method="POST">
  <table>
    <tr>
```

```
      <td colspan="4">
        <?php echo $form->renderGlobalErrors() ?>
      </td>
    </tr>

    // ...
  </table>
```

The call to the `renderGlobalErrors()` method displays the global error list. It is also possible to access the global errors using the `hasGlobalErrors()` and `getGlobalErrors()` methods, as shown in Listing 3-16.

Listing 3-16 - Global Errors customization with the `hasGlobalErrors()` and `getGlobalErrors()` Methods

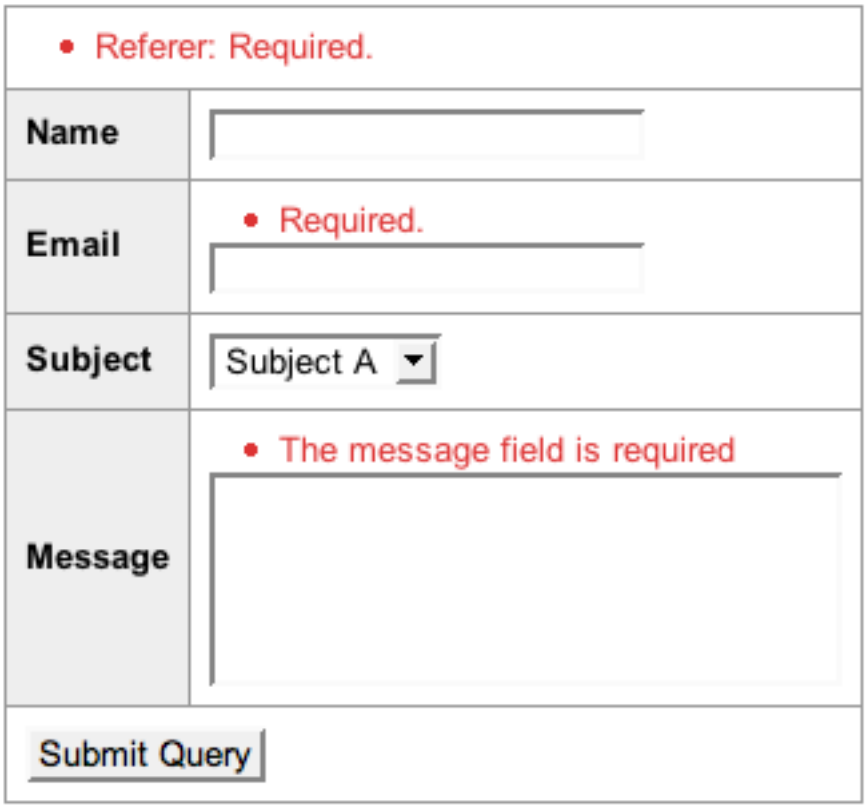
Listing 3-25

```
[php]
<?php if ($form->hasGlobalErrors()): ?>
  <tr>
    <td colspan="4">
      <ul class="error_list">
        <?php foreach ($form->getGlobalErrors() as $name => $error): ?>
          <li><?php echo $name.': '.$error ?></li>
        <?php endforeach; ?>
      </ul>
    </td>
  </tr>
<?php endif; ?>
```

Each global error has a name (`name`) and a message (`error`). The name is empty when there is a "real" global error, but when there is an error for a hidden field or a field that is not displayed, the `name` is the field label name.

Even if the template is now technically equivalent to the template we started with (Figure 3-8), the new one is now customizable.

Figure 3-8 - customized Form using the Field Methods



Internationalization

Every form element, such as labels and error messages, are automatically handled by the symfony internationalization system. This means that the web designer has nothing special to do if they want to internationalize forms, even when they explicitly override a label with the `renderLabel()` method. Translation is automatically taken into consideration. For further information about form internationalization, please see Chapter 9.

Interacting with the Developer

Let's end this chapter with a description of a typical form development scenario using symfony:

- The development team starts with implementing the form class and its action. The template is basically nothing more than the `<?php echo $form ?>` prototyping statement.
- In the meantime, designers design the style guidelines and the display rules that apply to the forms: global structure, error message displaying rules, ...
- Once the business logic is set and the style guidelines confirmed, the web designer team can modify the form templates and customize them. The team just need to know the name of the fields and the action required to handle the form's life cycle.

When this first cycle is over, both business rule modifications and template modifications can be done at the same time.

Without impacting the templates, therefore without any designer team intervention needed, the development team is able to:

- Modify the widgets of the form
- Customize error messages
- Edit, add, or delete validation rules

Likewise, the designer team is free to perform any ergonomic or graphic changes without falling back on the development team.

But the following actions involve coordination between the teams:

- Renaming a field
- Adding or deleting a field

This cooperation makes sense as it involves changes in both business rules and form display. Like we stated at the beginning of this chapter, even if the form system cleanly separates the tasks, there is nothing like communication between the teams.

Chapter 4

Propel Integration

In a Web project, most forms are used to create or modify model objects. These objects are usually serialized in a database thanks to an ORM. Symfony's form system offers an additional layer for interfacing with Propel, symfony's built-in ORM, making the implementation of forms based on these model objects easier.

This chapter goes into detail about how to integrate forms with Propel object models. It is highly suggested to be already acquainted with Propel and its integration in symfony. If this is not the case, refer to the chapter Inside the Model Layer[7] from the "The Definitive Guide to symfony" book.

Before we start

In this chapter, we will create an article management system. Let's start with the database schema. it is made of five tables: `article`, `author`, `category`, `tag`, and `article_tag`, as Listing 4-1 shows.

Listing 4-1 - Database Schema

Listing 4-1

```
// config/schema.yml
propel:
  article:
    id:           ~
    title:        { type: varchar(255), required: true }
    slug:         { type: varchar(255), required: true }
    content:      longvarchar
    status:       varchar(255)
    author_id:    { type: integer, required: true, foreignTable: author,
foreignReference: id, OnDelete: cascade }
    category_id:  { type: integer, required: false, foreignTable:
category, foreignReference: id, onDelete: setnull }
    published_at: timestamp
    created_at:   ~
    updated_at:   ~
    _uniques:
      unique_slug: [slug]

  author:
    id:           ~
    first_name:   varchar(20)
    last_name:    varchar(20)
```

7. http://www.symfony-project.org/book/1_1/08-Inside-the-Model-Layer

```
    email:        { type: varchar(255), required: true }
    active:       boolean

  category:
    id:           ~
    name:         { type: varchar(255), required: true }

  tag:
    id:           ~
    name:         { type: varchar(255), required: true }

  article_tag:
    article_id:   { type: integer, foreignTable: article,
foreignReference: id, primaryKey: true, onDelete: cascade }
    tag_id:       { type: integer, foreignTable: tag, foreignReference:
id, primaryKey: true, onDelete: cascade }
```

Here are the relations between the tables:

- 1-n relation between the `article` table and the `author` table: an article is written by one and only one author
- 1-n relation between the `article` table and the `category` table: an article belongs to one or zero category
- n-n relation between the `article` and `tag` tables

Generating Form Classes

We want to edit the information of the `article`, `author`, `category`, and `tag` tables. To do so, we need to create forms linked to each of these tables and configure widgets and validators related to the database schema. Even if it is possible to create these forms manually, it is a long, tedious task, and overall, it forces repetition of the same kind of information in several files (column and field name, maximum size of column and fields, ...). Furthermore, each time we change the model, we will also have to change the related form class. Fortunately, the Propel plugin has a built-in task `propel:build-forms` that automates this process generating the forms related to the object model:

Listing 4-2
```
$ ./symfony propel:build-forms
```

During the form generation, the task creates one class per table with validators and widgets for each column using introspection of the model and taking into account relations between tables.

The `propel:build-all` and `propel:build-all-load` also updates form classes, automatically invoking the `propel:build-forms` task.

After executing these tasks, a file structure is created in the `lib/form/` directory. Here are the files created for our example schema:

Listing 4-3
```
lib/
  form/
    BaseFormPropel.class.php
    ArticleForm.class.php
    ArticleTagForm.class.php
    AuthorForm.class.php
    CategoryForm.class.php
```

```
TagForm.class.php
base/
  BaseArticleForm.class.php
  BaseArticleTagForm.class.php
  BaseAuthorForm.class.php
  BaseCategoryForm.class.php
  BaseTagForm.class.php
```

The `propel:build-forms` task generates two classes for each table of the schema, one base class in the `lib/form/base` directory and one in the `lib/form/` directory. For example, the `author` table, consists of `BaseAuthorForm` and `AuthorForm` classes that were generated in the files `lib/form/base/BaseAuthorForm.class.php` and `lib/form/AuthorForm.class.php`.

Forms Generation Directory

The `propel:build-forms` task generates these files in a structure similar to the Propel structure. The `package` attribute of the Propel schema allows to logically put together tables subsets. The default package is `lib.model`, so Propel generates these files in the `lib/model/` directory and the forms are generated in the `lib/form` directory. Using the `lib.model.cms` package as shown in the example below, Propel classes will be generated in the `lib/model/cms/` directory and the form classes in the `lib/form/cms/` directory.

Listing 4-4

```
propel:
  _attributes: { noXsd: false, defaultIdMethod: none, package:
lib.model.cms }
  # ...
```

Packages are useful to split the database schema up and to deliver forms within a plugin as we will see in Chapter 5.

For further information on Propel packages, please refer to the Inside the Model Layer[8] chapter of "The Definitive Guide to symfony".

Table below sums up the hierarchy among the different classes involved in the `AuthorForm` form definition.

Class	Package	For	Description
AuthorForm	project	developer	Overrides generated form
BaseAuthorForm	project	symfony	Based on the schema and overridden at each execution of the `propel:build-forms` task
BaseFormPropel	project	developer	Allows global Customization of Propel forms
sfFormPropel	Propel plugin	symfony	Base of Propel forms
sfForm	symfony	symfony	Base of symfony forms

In order to create or edit an object from the `Author` class, we will use the `AuthorForm` class, described in Listing 4-2. As you can notice, this class does not contain any methods as it inherits from the `BaseAuthorForm` which is generated through the configuration. The `AuthorForm` class is the class we will use to Customize and override the form configuration.

Listing 4-2 - `AuthorForm` *Class*

Listing 4-5

```
class AuthorForm extends BaseAuthorForm
{
```

8. `http://www.symfony-project.org/book/1_1/08-Inside-the-Model-Layer`

```
  public function configure()
  {
  }
}
```

Listing 4-3 shows the `BaseAuthorForm` class with the validators and widgets generated introspecting the model for the `author` table.

Listing 4-3 - `BaseAuthorForm` *Class representing the Form for the* `author` *table*

Listing 4-6

```
class BaseAuthorForm extends BaseFormPropel
{
  public function setup()
  {
    $this->setWidgets(array(
      'id'         => new sfWidgetFormInputHidden(),
      'first_name' => new sfWidgetFormInput(),
      'last_name'  => new sfWidgetFormInput(),
      'email'      => new sfWidgetFormInput(),
    ));

    $this->setValidators(array(
      'id'         => new sfValidatorPropelChoice(array('model' =>
'Author', 'column' => 'id', 'required' => false)),
      'first_name' => new sfValidatorString(array('max_length' => 20,
'required' => false)),
      'last_name'  => new sfValidatorString(array('max_length' => 20,
'required' => false)),
      'email'      => new sfValidatorString(array('max_length' => 255)),
    ));

    $this->widgetSchema->setNameFormat('author[%s]');

    $this->errorSchema = new
sfValidatorErrorSchema($this->validatorSchema);

    parent::setup();
  }

  public function getModelName()
  {
    return 'Author';
  }
}
```

The generated class looks very similar to the forms we have already created in the previous chapters, except for a few things:

- The base class is `BaseFormPropel` instead of `sfForm`
- The validator and widget configuration takes place in the `setup()` method, rather than in the `configure()` method
- The `getModelName()` method returns the Propel class related to this form

Global Customization of Propel Forms

In addition to the classes generated for each table, the `propel:build-forms` also generates a `BaseFormPropel` class. This empty class is the base class of every other generated class in the `lib/form/base/` directory and allows to configure the behavior of every Propel form globally. For example, it is possible to easily change the default formatter for all Propel forms:

Listing 4-7

```
abstract class BaseFormPropel extends sfFormPropel
{
  public function setup()
  {
    sfWidgetFormSchema::setDefaultFormFormatterName('div');
  }
}
```

You'll notice that the `BaseFormPropel` class inherits from the `sfFormPropel` class. This class incorporates functionality specific to Propel and among other things deals with the object serialization in database from the values submitted in the form.

TIP Base classes use the `setup()` method for the configuration instead of the `configure()` method. This allows the developer to override the configuration of empty generated classes without handling the `parent::configure()` call.

The form field names are identical to the column names we set in the schema: `id`, `first_name`, `last_name`, and `email`.

For each column of the `author` table, the `propel:build-forms` task generates a widget and a validator according to the schema definition. The task always generates the most secure validators possible. Let's consider the `id` field. We could just check if the value is a valid integer. Instead the validator generated here allows us to also validate that the identifier actually exists (to edit an existing object) or that the identifier is empty (so that we could create a new object). This is a stronger validation.

The generated forms can be used immediately. Add a `<?php echo $form ?>` statement, and this will allow to create functional forms with validation **without writing a single line of code**.

Beyond the ability to quickly make prototypes, generated forms are easy to extend without having to modify the generated classes. This is thanks to the inheritance mechanism of the base and form classes.

At last at each evolution of the database schema, the task allows to generate again the forms to take into account the schema modifications, without overriding the Customization you might have made.

The CRUD Generator

Now that there are generated form classes, let's see how easy it is to create a symfony module to deal with the objects from a browser. We wish to create, modify, and delete objects from the `Article`, `Author`, `Category`, and `Tag` classes. Let's start with the module creation for the `Author` class. Even if we can manually create a module, the Propel plugin provides the `propel:generate-crud` task which generates a CRUD module based on a Propel object model class. Using the form we generated in the previous section:

Listing 4-8

```
$ ./symfony propel:generate-crud frontend author Author
```

The `propel:generate-crud` takes three arguments:

- `frontend` : name of the application you want to create the module in
- `author` : name of the module you want to create
- `Author` : name of the model class you want to create the module for

CRUD stands for Creation / Retrieval / Update / Deletion and sums up the four basic operations we can carry out with the model datas.

In Listing 4-4, we see that the task generated five actions allowing us to list (`index`), create (`create`), modify (`edit`), save (`update`), and delete (`delete`) the objects of the `Author` class.

Listing 4-4 - The `authorActions` *Class generated by the Task*

Listing 4-9

```
// apps/frontend/modules/author/actions/actions.class.php
class authorActions extends sfActions
{
  public function executeIndex()
  {
    $this->authorList = AuthorPeer::doSelect(new Criteria());
  }

  public function executeCreate()
  {
    $this->form = new AuthorForm();

    $this->setTemplate('edit');
  }

  public function executeEdit($request)
  {
    $this->form = new
AuthorForm(AuthorPeer::retrieveByPk($request->getParameter('id')));
  }

  public function executeUpdate($request)
  {
    $this->forward404Unless($request->isMethod('post'));

    $this->form = new
AuthorForm(AuthorPeer::retrieveByPk($request->getParameter('id')));

    $this->form->bind($request->getParameter('author'));
    if ($this->form->isValid())
    {
      $author = $this->form->save();

      $this->redirect('author/edit?id='.$author->getId());
    }

    $this->setTemplate('edit');
  }

  public function executeDelete($request)
  {
    $this->forward404Unless($author =
AuthorPeer::retrieveByPk($request->getParameter('id')));
```

```
    $author->delete();

    $this->redirect('author/index');
  }
}
```

In this module, the form life cycle is handled by three methods: `create`, `edit` and, `update`. It is also possible to ask the `propel:generate-crud` task to generate only one method covering the three previous methods functionalities, with the option `--non-atomic-actions`:

```
$ ./symfony propel:generate-crud frontend author Author
--non-atomic-actions
```

Listing 4-10

The generated code using `--non-atomic-actions` (Listing 4-5) is more concise and less verbose.

Listing 4-5 - The `authorActions` *Class generated with the* `--non-atomic-actions` *option*

```
class authorActions extends sfActions
{
  public function executeIndex()
  {
    $this->authorList = AuthorPeer::doSelect(new Criteria());
  }

  public function executeEdit($request)
  {
    $this->form = new
AuthorForm(AuthorPeer::retrieveByPk($request->getParameter('id')));

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('author'));
      if ($this->form->isValid())
      {
        $author = $this->form->save();

        $this->redirect('author/edit?id='.$author->getId());
      }
    }
  }

  public function executeDelete($request)
  {
    $this->forward404Unless($author =
AuthorPeer::retrieveByPk($request->getParameter('id')));

    $author->delete();

    $this->redirect('author/index');
  }
}
```

Listing 4-11

The task also generated two templates, `indexSuccess` and `editSuccess`. The `editSuccess` template was generated without using the `<?php echo $form ?>` statement. We can modify this behavior, using the `--non-verbose-templates`:

Listing 4-12

```
$ ./symfony propel:generate-crud frontend author Author
--non-verbose-templates
```

This option is helpful during prototyping phases, as Listing 4-6 shows.

Listing 4-6 - The `editSuccess` *Template*

Listing 4-13

```
// apps/frontend/modules/author/templates/editSuccess.php
<?php $author = $form->getObject() ?>
<h1><?php echo $author->isNew() ? 'New' : 'Edit' ?> Author</h1>

<form action="<?php echo url_for('author/edit'.(!$author->isNew() ?
'?id='.$author->getId() : '')) ?>" method="post" <?php
$form->isMultipart() and print 'enctype="multipart/form-data" ' ?>>
  <table>
    <tfoot>
      <tr>
        <td colspan="2">
           <a href="<?php echo url_for('author/index') ?>">Cancel</a>
          <?php if (!$author->isNew()): ?>
             <?php echo link_to('Delete', 'author/
delete?id='.$author->getId(), array('post' => true, 'confirm' => 'Are you
sure?')) ?>
          <?php endif; ?>
          <input type="submit" value="Save" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

The `--with-show` option let us generate an action and a template we can use to view an object (read only).

You can now open the URL `/frontend_dev.php/author` in a browser to view the generated module (Figure 4-1 and Figure 4-2). Take time to play with the interface. Thanks to the generated module you can list the authors, add a new one, edit, modify, and even delete. You will also notice that the validation rules are also working.

Figure 4-1 - Authors List

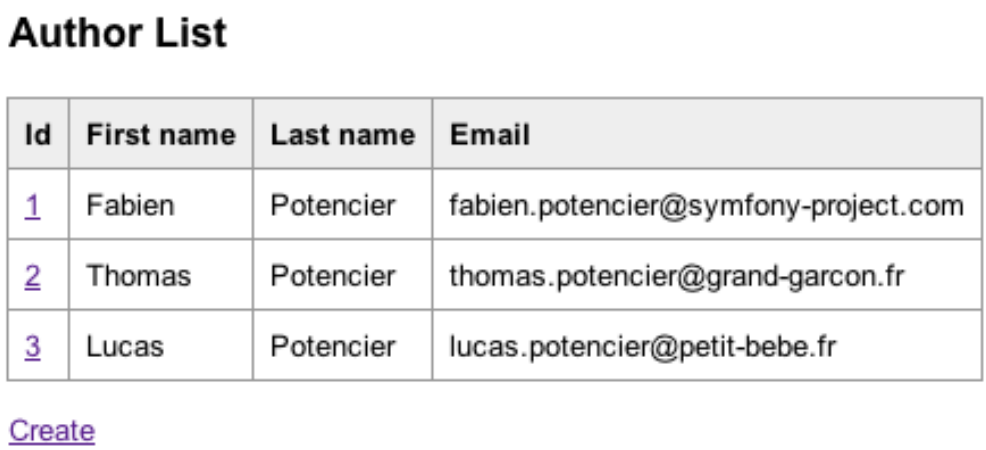
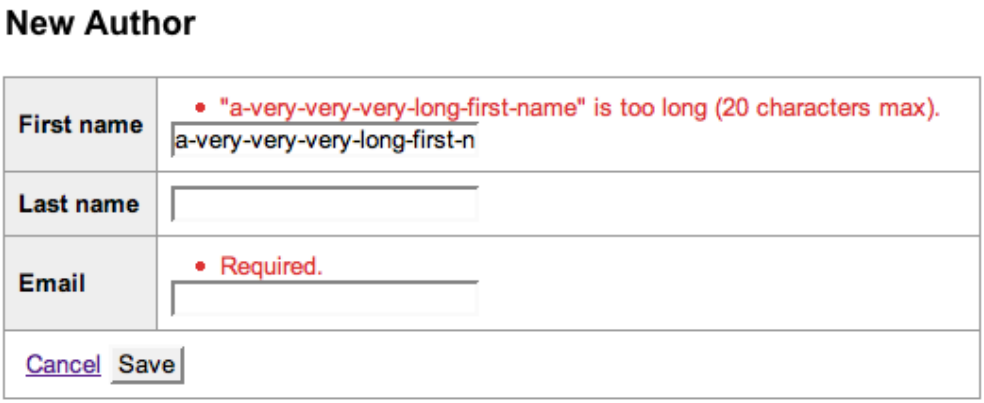


Figure 4-2 - Editing an Author with Validation Errors



We can now repeat the operation with the `Article` class:

```
$ ./symfony propel:generate-crud frontend article Article
--non-verbose-templates --non-atomic-actions
```

Listing 4-14

The generated code is quite similar to the code of the `Author` class. However, if you try to create a new article, the code throws a fatal error as you can see in Figure 4-3.

Figure 4-3 - Linked Tables must define the `__toString()` method

The `ArticleForm` form uses the `sfWidgetFormPropelSelect` widget to represent the relation between the `Article` object and the `Author` object. This widget creates a drop-down list with the authors. During the display, the authors objects are converted into a string of characters using the `__toString()` magic method, which must be defined in the `Author` class as shown in Listing 4-7.

Listing 4-7 - Implementing the `__toString()` method for the `Author` class

```
class Author extends BaseAuthor
{
  public function __toString()
  {
```

Listing 4-15

```
    return $this->getFirstName().' '.$this->getLastName();
  }
}
```

Just like the `Author` class, you can create `__toString()` methods for the other classes of our model: `Article`, `Category`, and `Tag`.

The `method` option of the `sfWidgetFormPropelSelect` widget change the method used to represent an object in text format.

The Figure 4-4 Shows how to create an article after having implemented the `__toString()` method.

Figure 4-4 - Creating an Article

Customizing the generated Forms

The `propel:build-forms` and `propel:generate-crud` tasks let us create functional symfony modules to list, create, edit, and delete model objects. These modules are taking into account

not only the validation rules of the model but also the relationships between tables. All of this happens without writing a single line of code!

The time has now come to customize the generated code. If the form classes are already considering many elements, some aspects will need to be customized.

Configuring validators and widgets

Let's start with configuring the validators and widgets generated by default.

The `ArticleForm` form has a `slug` field. The slug is a string of characters that uniquely representing the article in the URL. For instance, the slug of an article whose title is "Optimize the developments with symfony" is `12-optimize-the-developments-with-symfony`, `12` being the article `id`. This field is usually automatically computed when the object is saved, depending on the `title`, but it has the potential to be explicitly overridden by the user. Even if this field is required in the schema, it can not be compulsory to the form. That is why we modify the validator and make it optional, as in Listing 4-8. We will also customize the `content` field increasing its size and forcing the user to type in at least five characters.

Listing 4-8 - Customizing Validators and Widgets

Listing 4-16

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $this->validatorSchema['slug']->setOption('required', false);
    $this->validatorSchema['content']->setOption('min_length', 5);

    $this->widgetSchema['content']->setAttributes(array('rows' => 10,
'cols' => 40));
  }
}
```

We use here the `validatorSchema` and `widgetSchema` objects as PHP arrays. These arrays are taking the name of a field as key and return respectively the validator object and the related widget object. We can then Customize individually fields and widgets.

In order to allow the use of objects as PHP arrays, the `sfValidatorSchema` and `sfWidgetFormSchema` classes implement the `ArrayAccess` interface, available in PHP since version 5.

To make sure two articles can not have the same `slug`, a uniqueness constraint has been added in the schema definition. This constraint on the database level is reflected in the `ArticleForm` form using the `sfValidatorPropelUnique` validator. This validator can check the uniqueness of any form field. It is helpful among other things to check the uniqueness of an email address of a login for instance. Listing 4-9 shows how to use it in the `ArticleForm` form.

Listing 4-9 - Using the `sfValidatorPropelUnique` validator to check the Uniqueness of a field

Listing 4-17

```
class BaseArticleForm extends BaseFormPropel
{
  public function setup()
  {
    // ...

    $this->validatorSchema->setPostValidator(
```

```
      new sfValidatorPropelUnique(array('model' => 'Article', 'column' =>
array('slug')))
    );
  }
}
```

The `sfValidatorPropelUnique` validator is a `postValidator` running on the whole data after the individual validation of each field. In order to validate the `slug` uniqueness, the validator must be able to access, not only the `slug` value, but also the value of the primary key(s). Validation rules are indeed different throughout the creation and the edition since the slug can stay the same during the update of an article.

Let's Customize now the `active` field of the `author` table, used to know if an author is active. Listing 4-10 shows how to exclude inactive authors from the `ArticleForm` form, modifying the `criteria` option of the `sfWidgetPropelSelect` widget connected to the `author_id` field. The `criteria` option accepts a Propel Criteria object, allowing to narrow down the list of available options in the rolling list.

Listing 4-10 - Customizing the `sfWidgetPropelSelect` *widget*

Listing 4-18
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $authorCriteria = new Criteria();
    $authorCriteria->add(AuthorPeer::ACTIVE, true);

    $this->widgetSchema['author_id']->setOption('criteria',
$authorCriteria);
  }
}
```

Even if the widget customization can make us narrow down the list of available options, we must not forget to consider this narrowing on the validator level, as shown in Listing 4-11. Like the `sfWidgetProperSelect` widget, the `sfValidatorPropelChoice` validator accepts a `criteria` option to narrow down the options valid for a field.

Listing 4-11 - Customizing the `sfValidatorPropelChoice` *validator*

Listing 4-19
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $authorCriteria = new Criteria();
    $authorCriteria->add(AuthorPeer::ACTIVE, true);

    $this->widgetSchema['author_id']->setOption('criteria',
$authorCriteria);
    $this->validatorSchema['author_id']->setOption('criteria',
$authorCriteria);
  }
}
```

In the previous example we defined the `Criteria` object directly in the `configure()` method. In our project, this criteria will certainly be helpful in other circumstances, so it is better to

create a `getActiveAuthorsCriteria()` method within the `AuthorPeer` class and to call this method from `ArticleForm` as Listing 4-12 shows.

Listing 4-12 - Refactoring the `Criteria` in the Model

Listing 4-20

```
class AuthorPeer extends BaseAuthorPeer
{
  static public function getActiveAuthorsCriteria()
  {
    $criteria = new Criteria();
    $criteria->add(AuthorPeer::ACTIVE, true);

    return $criteria;
  }
}

class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $authorCriteria = AuthorPeer::getActiveAuthorsCriteria();
    $this->widgetSchema['author_id']->setOption('criteria',
$authorCriteria);
    $this->validatorSchema['author_id']->setOption('criteria',
$authorCriteria);
  }
}
```

Like the `sfWidgetPropelSelect` widget and the `sfValidatorPropelChoice` validator represent a 1-n relation between two tables, the `sfWidgetFormPropelSelectMany` and the `sfValidatorPropelChoiceMany` validator represent a n-n relation and accept the same options. In the `ArticleForm` form, these classes are used to represent a relation between the `article` table and the `tag` table.

Changing validator

The `email` being defined as a `varchar(255)` in the schema, symfony created a `sfValidatorString()` validator restraining the maximum length to 255 characters. This field is also supposed to receive a valid email, Listing 4-14 replaces the generated validator with a `sfValidatorEmail` validator.

Listing 4-13 - Changing the `email` field Validator of the `AuthorForm` class

Listing 4-21

```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorEmail();
  }
}
```

Adding a validator

We observed in the previous chapter how to modify the generated validator. But in the case of the `email` field, it would be useful to keep the maximum length validation. In Listing 4-14, we use the `sfValidatorAnd` validator to guarantee the email validity and check the maximum length allowed for the field.

Listing 4-14 - Using a multiple Validator

Listing 4-22
```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      new sfValidatorString(array('max_length' => 255)),
      new sfValidatorEmail(),
    ));
  }
}
```

The previous example is not perfect, because if we decide later to modify the length of the `email` field in the database schema, we will have to think about doing it also in the form. Instead of replacing the generated validator, it is better to add one, as shown in Listing 4-15.

Listing 4-15 - Adding a Validator

Listing 4-23
```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));
  }
}
```

Changing widget

In the database schema, the `status` field of the `article` table stores the article status as a string of characters. The possible values were defined in the `ArticePeer` class, as shown in Listing 4-16.

Listing 4-16 - Defining available Statuses in the `ArticlePeer` class

Listing 4-24
```
class ArticlePeer extends BaseArticlePeer
{
  static protected $statuses = array('draft', 'online', 'offline');

  static public function getStatuses()
  {
    return self::$statuses;
  }

  // ...
}
```

When editing an article, the `status` field must be represented as a drop-down list instead of a text field. To do so, let's change the widget we used, as shown in Listing 4-17.

Listing 4-17 - Changing the Widget for the `status` field

Listing 4-25
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
```

```
    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
=> ArticlePeer::getStatuses()));
  }
}
```

To be thorough we must also change the validator to make sure the chosen status actually belongs to the list of possible options (Listing 4-18).

Listing 4-18 - Modifying the `status` *Field Validator*

```
class ArticleForm extends BaseArticleForm                                Listing 4-26
{
  public function configure()
  {
    $statuses = ArticlePeer::getStatuses();

    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
=> $statuses));

    $this->validatorSchema['status'] = new
sfValidatorChoice(array('choices' => array_keys($statuses)));
  }
}
```

Deleting a field

The `article` table has two special columns, `created_at` and `updated_at`, whose update is automatically handled by Propel. We must then delete them from the form as Listing 4-19 show, to prevent the user from modifying them.

Listing 4-19 - Deleting a Field

```
class ArticleForm extends BaseArticleForm                                Listing 4-27
{
  public function configure()
  {
    unset($this->validatorSchema['created_at']);
    unset($this->widgetSchema['created_at']);

    unset($this->validatorSchema['updated_at']);
    unset($this->widgetSchema['updated_at']);
  }
}
```

In order to delete a field, it is necessary to delete its validator and its widget. Listing 4-20 shows how it is also possible to delete both in one action, using the form as a PHP array.

Listing 4-20 - Deleting a Field using the Form as a PHP Array

```
class ArticleForm extends BaseArticleForm                                Listing 4-28
{
  public function configure()
  {
    unset($this['created_at'], $this['updated_at']);
  }
}
```

Sum up

To sum up, Listing 4-21 and Listing 4-22 show the `ArticleForm` and `AuthorForm` forms as we customize them.

Listing 4-21 - `ArticleForm` *Form*

Listing 4-29
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $authorCriteria = AuthorPeer::getActiveAuthorsCriteria();

    // widgets
    $this->widgetSchema['content']->setAttributes(array('rows' => 10,
'cols' => 40));
    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
=> ArticlePeer::getStatuses()));
    $this->widgetSchema['author_id']->setOption('criteria',
$authorCriteria);

    // validators
    $this->validatorSchema['slug']->setOption('required', false);
    $this->validatorSchema['content']->setOption('min_length', 5);
    $this->validatorSchema['status'] = new
sfValidatorChoice(array('choices' =>
array_keys(ArticlePeer::getStatuses())));
    $this->validatorSchema['author_id']->setOption('criteria',
$authorCriteria);

    unset($this['created_at']);
    unset($this['updated_at']);
  }
}
```

Listing 4-22 - `AuthorForm` *Form*

Listing 4-30
```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));
  }
}
```

Using the `propel:build-forms` allows to automatically generate most of the elements letting forms introspect the object model. This automatization is helpful for several reasons:

- It makes the developer's life easier, saving him from a repetitive and redundant work. He can then focus on the validators and widget Customization according to the project's specific business rules .
- Besides, when the database schema is updated, the generated forms will be automatically updated. The developer will just have to tune the customization they made.

The next section will describe the customization of actions and templates generated by the `propel:generate-crud` task.

Form Serialization

The previous section show us how to customize forms generated by the task `propel:build-forms`. In the current section, we will customize the life cycle of forms, starting from the code generated by the `propel:generate-crud` task.

Default values

A Propel form instance is always connected to a Propel object. The linked Propel object always belongs to the class returned by the `getModelName()` method. For instance, the `AuthorForm` form can only be linked to objects belonging to the `Author` class. This object is either an empty object (a blank instance of the `Author` class), or the object sent to the constructor as first argument. Whereas the constructor of an "average" form takes an array of values as first argument, the constructor of a Propel form takes a Propel object. This object is used to define each form field default value. The `getObject()` method returns the object related to the current instance and the `isNew()` method allows to know if the object was sent via the constructor:

Listing 4-31

```
// creating a new object
$authorForm = new AuthorForm();

print $authorForm->getObject()->getId(); // outputs null
print $authorForm->isNew();              // outputs true

// modifying an existing object
$author = AuthorPeer::retrieveByPk(1);
$authorForm = new AuthorForm($author);

print $authorForm->getObject()->getId(); // outputs 1
print $authorForm->isNew();              // outputs false
```

Handling life cycle

As we observed at the beginning of the chapter, the `edit` action, shown in Listing 4-23, handles the form life cycle.

Listing 4-23 - The `executeEdit` *Method of the* `author` *Module*

Listing 4-32

```
// apps/frontend/modules/author/actions/actions.class.php
class authorActions extends sfActions
{
  // ...

  public function executeEdit($request)
  {
    $author = AuthorPeer::retrieveByPk($request->getParameter('id'));
    $this->form = new AuthorForm($author);

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('author'));
      if ($this->form->isValid())
      {
```

```
        $author = $this->form->save();

        $this->redirect('author/edit?id='.$author->getId());
      }
    }
  }
}
```

Even if the `edit` action looks like the actions we might have describe in the previous chapters, we can point a few differences:

- A Propel object from the `Author` class is sent as first argument to the form constructor:

Listing 4-33
```
$author = AuthorPeer::retrieveByPk($request->getParameter('id'));
$this->form = new AuthorForm($author);
```

- The widgets `name` attribute format is automatically customize to allow the retrieval of the input data in a PHP array named after the related table (`author`):

Listing 4-34
```
$this->form->bind($request->getParameter('author'));
```

- When the form is valid, a mere call to the `save()` method creates or updates the Propel object related to the form:

Listing 4-35
```
$author = $this->form->save();
```

Creating and Modifying a Propel Object

Listing 4-23 code handles with a single method the creation and modification of objects from the `Author` class:

- Creation of a new `Author` object:
  - The `edit` action is called with no `id` parameter (`$request->getParameter('id')` is `null`)
  - The call to the `retrieveByPk()` therefore sends `null`
  - The `form` object is then linked to an empty `Author` Propel object
  - The `$this->form->save()` call creates consequently a new `Author` object when a valid form is submitted
- Modification of an existing `Author` object:
  - The `edit` action is called with an `id` parameter (`$request->getParameter('id')` standing for the primary key the `Author` object is to modify)
  - The call to the `retriveByPk()` method returns the `Author` object related to the primary key
  - The `form` object is therefore linked to the previously found object
  - The `$this->form->save()` call updates the `Author` object when a valid form is submitted

The `save()` method

When a Propel form is valid, the `save()` method updates the related object and stores it in the database. This method actually stores not only the main object but also the potentially related objects. For instance, the `ArticleForm` form updates the tags connected to an article. The relation between the `article` table and the `tag` table being a n-n relation, the tags related to an article are saved in the `article_tag` table (using the `saveArticleTagList()` generated method).

We will see in Chapter 9 that the `save()` method also automatically updates the internationalized tables.

In order to certify a consistent serialization, the `save()` method includes every updates in one transaction.

Using the `bindAndSave()` method

The `bindAndSave()` method binds the input data the user submitted to the form, validates this form and updates the related object in the database, all in one operation:

Listing 4-36

```
class articleActions extends sfActions
{
  public function executeCreate(sfWebRequest $request)
  {
    $this->form = new ArticleForm();

    if ($request->isMethod('post') &&
$this->form->bindAndSave($request->getParameter('article')))
    {
      $this->redirect('article/created');
    }
  }
}
```

Handling the files upload

The `save()` method automatically updates the Propel objects but can not handle the side elements as managing the file upload.

Let's see how to attach a file to each article. Files are stored in the `web/uploads` directory and a reference to the file path is kept in the `file` field of the `article` table, as shown in Listing 4-24.

Listing 4-24 - Schema for the `article` Table with associated File

Listing 4-37

```
// config/schema.yml
propel:
  article:
    // ...
    file: varchar(255)
```

After every schema update, you need to update the object model, the database and the related forms:

Listing 4-38

```
$ ./symfony propel:build-all
```

Do mind that the `propel:build-all` task deletes every schema tables to re-create them. The data inside the tables are therefore overwritten. That is why it is important to create test data (`fixtures`) you can download again at each model modification.

Listing 4-25 shows how to modify the `ArticleForm` class in order to link a widget and a validator to the `file` field.

Listing 4-25 - Modifying the `file` *Field of the* `ArticleForm` *form.*

Listing 4-39
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $this->widgetSchema['file'] = new sfWidgetFormInputFile();
    $this->validatorSchema['file'] = new sfValidatorFile();
  }
}
```

As for every form allowing to upload a file, does not forget to add also the `enctype` attribute to the `form` tag of the template (see Chapter 2 for further informations concerning file upload management).

Listing 4-26 shows the modifications to apply when saving the form to upload the file onto the server and store its path in the `article` object.

Listing 4-26 - Saving the `article` *Object and the File uploaded in the Action*

Listing 4-40
```
public function executeEdit($request)
{
  $author = ArticlePeer::retrieveByPk($request->getParameter('id'));
  $this->form = new ArticleForm($author);

  if ($request->isMethod('post'))
  {
    $this->form->bind($request->getParameter('article'),
$request->getFiles('article'));
    if ($this->form->isValid())
    {
      $file = $this->form->getValue('file');
      $filename =
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
      $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

      $article = $this->form->save();

      $this->redirect('article/edit?id='.$article->getId());
    }
  }
}
```

Saving the uploaded file on the filesystem allows the `sfValidatedFile` object to know the absolute path to the file. During the call to the `save()` method, the fields values are used to update the related object and, as for the `file` field, the `sfValidatedFile` object is converted in a character string thanks to the `__toString()` method, sending back the absolute path to the file. The `file` column of the `article` table will store this absolute path.

If you wish to store the path relative to the `sfConfig::get('sf_upload_dir')` directory, you can create a class inheriting from `sfValidatedFile` and use the `validated_file_class` option to send to the `sfValidatorFile` validator the name of the new class. The validator will then return an instance of your class. We will see in the rest of this chapter another approach, consisting in modifying the value of the `file` column before saving the object in database.

Customizing the `save()` method

We observed in the previous section how to save the uploaded file in the `edit` action. One of the principles of the object oriented programming is the reusability of the code, thanks to its encapsulation in classes. Instead of duplicating the code used to save the file in each action using the `ArticleForm` form, it is better to move it in the `ArticleForm` class. Listing 4-27 shows how to override the `save()` method in order to also save the file and possibly to delete of an existing file.

Listing 4-27 - Overriding the `save()` Method of the `ArticleForm` Class

Listing 4-41

```
class ArticleForm extends BaseFormPropel
{
  // ...

  public function save(PropelPDO $con = null)
  {
    if (file_exists($this->getObject()->getFile()))
    {
      unlink($this->getObject()->getFile());
    }

    $file = $this->getValue('file');
    $filename =
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
    $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

    return parent::save($con);
  }
}
```

After moving the code to the form, the `edit` action is identical to the code initially generated by the `propel:generate-crud` task.

Refactoring the Code in the Model of in the Form

The actions generated by the `propel:generate-crud` task shouldn't usually be modified.

The logic you could add in the `edit` action, especially during the form serialization, must usually be moved in the model classes or in the form class.

We just went over an example of refactoring in the form class in order to consider a uploaded file storing. Let's take another example related to the model. The `ArticleForm` form has a `slug` field. We observed that this field should be automatically computed from the `title` field name that it should be potentially overridden by the user. This logic does not depend on the form. It belongs therefore to the model, as shown the following code:

Listing 4-42
```
class Article extends BaseArticle
{
  public function save(PropelPDO $con = null)
  {
    if (!$this->getSlug())
    {
      $this->setSlugFromTitle();
    }

    return parent::save($con);
  }

  protection function setSlugFromTitle()
  {
    // ...
  }
}
```

The main goal of those refactorings is to respect the separation in applicative layers, and especially the reusability of the developments.

Customizing the `doSave()` method

We observed that the saving of an object was made within a transaction in order to guarantee that each operation related to the saving is processed correctly. When overriding the `save()`method as we did in the previous section in order to save the uploaded file, the executed code is independent from this transaction.

Listing 4-28 shows how to use the `doSave()` method to insert in the global transaction our code saving the uploaded file.

Listing 4-28 - Overriding the `doSave()` *Method in the* `ArticleForm` *Form*

Listing 4-43
```
class ArticleForm extends BaseFormPropel
{
  // ...

  public function doSave($con = null)
  {
    if (file_exists($this->getObject()->getFile()))
    {
      unlink($this->getObject()->getFile());
    }

    $file = $this->getValue('file');
    $filename =
```

```
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
    $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

    return parent::doSave($con);
  }
}
```

The `doSave()` method being called in the transaction created by the `save()` method, if the call to the `save()` method of the `file()` object throws an exception, the object will not be saved.

Customizing the `updateObject()` Method

It is sometimes necessary to modify the object connected to the form between the update and the saving in database.

In our file upload example, instead of storing the absolute path to the uploaded file in the `file` column, we wish to store the path relative to the `sfConfig::get('sf_upload_dir')` directory.

Listing 4-29 shows how to override the `updateObject()` method of the `ArticleForm` form in order to change the value of the `file` column after the automatic update object but before it is saved.

Listing 4-29 - Overriding the `updateObject()` *Method and the* `ArticleForm` *Class*

Listing 4-44

```
class ArticleForm extends BaseFormPropel
{
  // ...

  public function updateObject()
  {
    $object = parent::updateObject();

    $object->setFile(str_replace(sfConfig::get('sf_upload_dir').'/', '',
$object->getFile()));

    return $object;
  }
}
```

The `updateObject()` method is called by the `doSave()` method before saving the object in database.

Chapter 5

Internationalization and Localization

A lot of popular Web applications are available in several languages, and sometimes, they are even customized based on the user culture. Symfony comes with a built-in framework that eases the management of these features (see chapter "I18n And L10n[9]" (`http://www.symfony-project.org/book/1_1/13-I18n-and-L10n`) of the symfony book).

The form framework also comes with built-in support for the user interface translation and provides an easy way to manage internationalized objects.

Form Internationalization

A symfony form is internationalizable by default. The translation of the **labels**, the **help texts**, and the **errors messages** can be done by editing the translation files, be they in the `XLIFF`, `gettext`, or any other symfony supported format.

Listing 8-1 shows the contact form we have developed in the previous chapters.

Listing 8-1 - Contact Form

Listing 5-1
```
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'name'  => new sfWidgetFormInput(),    // the default label is "Name"
      'email' => new sfWidgetFormInput(),    // the default label is
"Email"
      'body'  => new sfWidgetFormTextarea(), // the default label is "Body"
    ));

    // Change the email widget label
    $this->widgetSchema->setLabel('email', 'Email address');
  }
}
```

We can now define the label translations in the `XLIFF` file as shown in Listing 8-2 for the french language.

Listing 8-2 - XLIFF translation file

Listing 5-2
```
// apps/frontend/i18n/messages.fr.xml
<?xml version="1.0" ?>
```

9. `http://www.symfony-project.org/book/1_1/13-I18n-and-L10n`

```
<xliff version="1.0">
  <file original="global" source-language="en" datatype="plaintext">
    <body>
      <trans-unit>
        <source>Name</source>
        <target>Nom</target>
      </trans-unit>
      <trans-unit>
        <source>Email address</source>
        <target>Adresse email</target>
      </trans-unit>
      <trans-unit>
        <source>Body</source>
        <target>Message</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Specify the catalogue to use for translations

If you use the catalogue feature of the symfony i18n framework (`http://www.symfony-project.org/book/1_1/13-I18n-and-L10n#Managing%20Dictionaries`), you can bind a form to a given catalogue. In Listing 8-3, we associate the `ContactForm` form with the `contact_form` catalogue. So, the form element translations will be looked for in the `contact_form.fr.xml` file.

Listing 8-3 - Translation Catalogue Customization

```
class ContactForm extends sfForm                                    Listing 5-3
{
  public function configure()
  {
    // ...


$this->widgetSchema->getFormFormatter()->setTranslationCatalogue('contact_form');
  }
}
```

The usage of catalogues allows a better organization of your translations by using one file per form for example.

Error Messages Internationalization

Sometimes, the error messages embed the value submitted by the user (for example, "The email address user@domain is not valid."). We have already seen in Chapter 2 that this can be done easily in the form class by defining customized error messages and using references to the user submitted values. These references follow the `%parameter_name%` pattern.

The Listing 8-4 shows how to apply this principle to the `name` field of the contact form.

Listing 8-4 - Error Messages Internationalization

```
class ContactForm extends sfForm                                    Listing 5-4
{
  public function configure()
```

```
  {
    // ...

    $this->validatorSchema['name'] = new sfValidatorEmail(
      array('min_length' => 2, 'max_length' => 45),
      array('min_length' => 'Name "%value%" must be at least %min_length%
characters.',
            'max_length' => 'Name "%value%" must not exceed %max_length%
characters.',
      ),
    );
  }
}
```

We can now translate these error messages by editing the XLIFF file as shown in Listing 8-5.

Listing 8-5 - XLIFF Translation File for Error Messages

Listing 5-5

```
<trans-unit>
  <source>Name "%value%" must be at least %min_length% characters</source>
  <target>Le nom "%value%" doit comporter un minimum de %min_length%
caractères</target>
</trans-unit>
<trans-unit>
  <source>Name "%value%" must not exceed %max_length% characters</source>
  <target>Le nom "%value%" ne peut comporter plus de %max_length%
caractères</target>
</trans-unit>
```

Customization of the Translation object

If you want to use the symfony form framework without the symfony i18n framework, you need to provide your own *translation object*.

A translation object is just a *callable PHP*. It can be one of the following three things:

- a string representing a function name, like `my_function`
- an array with a reference to a class instance and the name of one of its methods, like `array($anObject, 'oneOfItsMethodsName')`
- a `sfCallable` instance. This class encapsulate a PHP callable in a consistent way.

A PHP callable is a reference to a function or a method instance. It is also a PHP variable that returns `true` when passed to the `is_callable()` function.

Let's take an example. You have to migrate a project which already has its own internationalization mechanism provided by the class show in Listing 8-6.

Listing 8-6 - Custom I18N class

Listing 5-6

```
class myI18n
{
  static protected $default_culture = 'en';
  static protected $messages = array('fr' => array(
    'Name'    => 'Nom',
    'Email'   => 'Courrier électronique',
    'Subject' => 'Sujet',
```

```
    'Body'    => 'Message',
  ));

  static public function translateText($text)
  {
    $culture = isset($_SESSION['culture']) ? $_SESSION['culture'] :
self::$default_culture;
    if (array_key_exists($culture, self::$messages)
        && array_key_exists($text, self::$messages[$culture]))
    {
      return self::$messages[$_SESSION['culture']][$text];
    }
    return $text;
  }
}

// Class usage
$myI18n = new myI18n();

$_SESSION['culture'] = 'en';
echo $myI18n->translateText('Subject'); // => display "Subject"

$_SESSION['culture'] = 'fr';
echo $myI18n->translateText('Subject'); // => display "Sujet"
```

Each form can define its very own callable which will manage the internationalization of the form elements as shown in Listing 8-7.

Listing 8-7 - Overriding of the Internationalization Method for a Form

```
class ContactForm extends sfForm                                    Listing 5-7
{
  public function configure()
  {
    // ...

$this->widgetSchema->getFormFormatter()->setTranslationCallable(array(new
myI18n(), 'translateText'));
  }
}
```

Translation Callable Accepted Parameters

The translation callable can take up to three arguments :

- the **text to translate**;
- an **associative array** of arguments to replace within the original text, typically to replace dynamic arguments as we have seen previously in this chapter;
- a **catalogue name** to use when translating the text.

Here is the call used by the `sfFormWidgetSchemaFormatter::translate()` method to call the translation callable:

```
return call_user_func(self::$translationCallable, $subject, $parameters,    Listing 5-8
$catalogue);
```

The `self::$translationCallable` is the reference to the translation callable. So, the previous code is equivalent to:

Listing 5-9
```
$myI18n->translateText($subject, $parameters, $catalogue);
```

Here is the updated version of the `MyI18n` class that supports those extra arguments:

Listing 5-10
```
class myI18n
{
  static protected $default_culture = 'en';
  static protected $messages = array('fr' => array(
    'messages' => array(
      'Name'    => 'Nom',
      'Email'   => 'Courrier électronique',
      'Subject' => 'Sujet',
      'Body'    => 'Message',
    ),
  ));

  static public function translateText($text, $arguments = array(),
$catalogue = 'messages')
  {
    $culture = isset($_SESSION['culture']) ? $_SESSION['culture'] :
self::$default_culture;
    if (array_key_exists($culture, self::$messages) &&
        array_key_exists($messages, self::$messages[$culture] &&
        array_key_exists($text, self::$messages[$culture][$messages]))
    {
      $text = self::$messages[$_SESSION['culture']][$messages][$text];
      $text = strtr($text, $arguments);
    }
    return $text;
  }
}
```

Why do we use the `sfWidgetFormSchemaFormatter` to customize the Translation Process?

As we have seen in Chapter 2, the form framework is based on the MVC architecture and the `sfWidgetFormSchemaFormatter` class belongs to the View layer. This class is responsible for all the text rendering, so it can intercept all the text strings and translate them on the fly.

Propel Objects Internationalization

The form framework has built-in support for Propel objects that are internationalized. Let's take an internationalized model example to illustrate the way it works:

Listing 5-11
```
propel:
  article:
    id:
    author:     varchar(255)
    created_at:
  article_i18n:
    title:      varchar(255)
    content:    longvarchar
```

You can generate the Propel classes and the related form classes with the following commands:

```
$ php symfony build:model
$ php symfony build:forms
```

Listing 5-12

Those commands generate some files in your symfony project:

```
lib/
  form/
    ArticleForm.class.php
    ArticleI18nForm.class.php
    BaseFormPropel.class.php
  model/
    Article.php
    ArticlePeer.php
    ArticleI18n.php
    ArticleI18nPeer.php
```

Listing 5-13

Listing 8-8 shows how to configure the `ArticleForm` to be able to edit the French and the English version of the article in the same form.

Listing 8-8 - I18n forms for an internationalized Propel Object

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $this->embedI18n(array('en', 'fr'));
  }
}
```

Listing 5-14

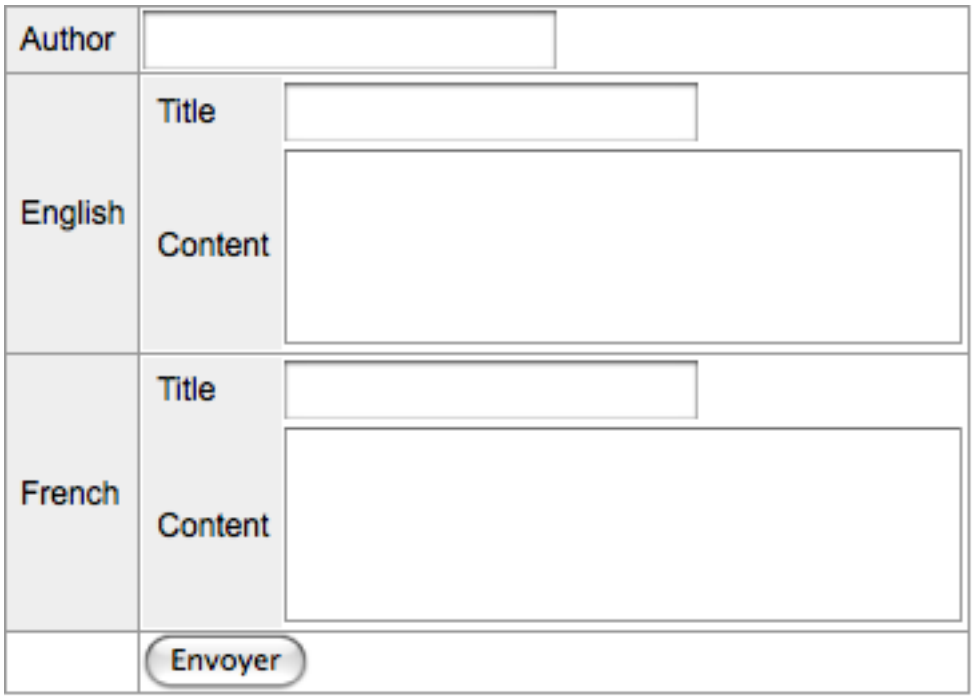
You can also customize the language labels of the form by adding the following code to the `configure()` method as shown in Listing 8-9.

Listing 8-9 - Language Labels Customizations

```
$this->widgetSchema->setLabel('en', 'English');
$this->widgetSchema->setLabel('fr', 'French');
```

Listing 5-15

Figure 8-1 - Internationalized Propel Form



That's all there is to it. When you call the `save()` method of the form object, the associated Propel object and all the i18n objects are saved automatically.

How to pass the User Culture to a Form?

If you want to bind a form to the current culture of the user, you can pass an optional `culture` option when you create the form:

Listing 5-16

```
class articleActions extends sfActions
{
  public function executeCreate($request)
  {
    $this->form = new ArticleForm(null, array('culture' =>
$this->getUser()->getCulture()));

    if ($request->isMethod('post') &&
$this->form->bindAndSave($request->getParameter('article')))
    {
      $this->redirect('article/created');
    }
  }
}
```

In the `ArticleForm` class, you can now get the value from the `options` array:

Listing 5-17

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $this->embedI18n(array($this->getCurrentCulture()));
  }

  public function getCurrentCulture()
  {
    return isset($this->options['culture']) ? $this->options['culture'] :
'en';
  }
}
```

Localized Widgets

The symfony form framework is bundled with some widgets that are i18n "aware". They can be used to localize some widgets according to the user culture.

Dates selectors

Here are the available widgets to localize a date:

- The `sfWidgetFormI18nDate` widget displays inputs for a date (day, month, year):

  Listing 5-18

  ```
  $this->widgetSchema['published_on'] = new
  sfWidgetFormI18nDate(array('culture' => 'fr'));
  ```

  You can also define the display format of the month, thanks to the `month_format` option which can take three different values:

  - `name` to display the name of the month (the default)
  - `short_name` to display the abbreviated name of the month

    - `number` to display the number of the month (from 1 to 12)

- The `sfWidgetFormI18nTime` widget displays input for a time (hours, minutes, and seconds):

Listing 5-19
```
$this->widgetSchema['published_on'] = new
sfWidgetFormI18nTime(array('culture' => 'fr'));
```

- The `sfWidgetFormI18nDateTime` widget displays inputs for a date and a time:

Listing 5-20
```
$this->widgetSchema['published_on'] = new
sfWidgetFormI18nDateTime(array('culture' => 'fr'));
```

Country selector

The `sfWidgetFormI18nSelectCountry` widget displays a select box filled with a list of countries. The country names are translated in the given language:

Listing 5-21
```
$this->widgetSchema['country'] = new
sfWidgetFormI18nSelectCountry(array('culture' => 'fr'));
```

You can also restrict the countries in the select box, thanks to the `countries` option:

Listing 5-22
```
$countries = array('fr', 'en', 'es', 'de', 'nl');
$this->widgetSchema['country'] = new
sfWidgetFormI18nSelectCountry(array('culture'   => 'fr',

'countries' => $countries));
```

Culture selector

The `sfWidgetFormI18nSelectLanguage` widget displays a select box filled with a list of languages. The language names are translated in the given language:

Listing 5-23
```
$this->widgetSchema['language'] = new
sfWidgetFormI18nSelectLanguage(array('culture' => 'fr'));
```

You can also restrict the languages in the select box, thanks to the `languages` option:

Listing 5-24
```
$languages = array('fr', 'en', 'es', 'de', 'nl');
$this->widgetSchema['language'] = new
sfWidgetFormI18nSelectLanguage(array('culture'   => 'fr',

'languages' => $languages));
```

Chapter 6

Doctrine Integration

In a Web project, most forms are used to create or modify model objects. These objects are usually serialized in a database thanks to an ORM. Symfony's form system offers an additional layer for interfacing with Doctrine, symfony's built-in ORM, making the implementation of forms based on these model objects easier.

This chapter goes into detail about how to integrate forms with Doctrine object models. It is highly suggested to be already acquainted with Doctrine and its integration in symfony. If this is not the case, refer to the chapter Inside the Model Layer[10] from the "The Definitive Guide to symfony" book.

Before we start

In this chapter, we will create an article management system. Let's start with the database schema. It is made of five tables: `article`, `author`, `category`, `tag`, and `article_tag`, as Listing 4-1 shows.

Listing 4-1 - Database Schema

Listing 6-1

```
// config/doctrine/schema.yml
Article:
  actAs: [Sluggable, Timestampable]
  columns:
    title:
      type: string(255)
      notnull: true
    content:
      type: clob
    status: string(255)
    author_id: integer
    category_id: integer
    published_at: timestamp
  relations:
    Author:
      foreignAlias: Articles
    Category:
      foreignAlias: Articles
    Tags:
      class: Tag
      refClass: ArticleTag
      foreignAlias: Articles
```

10. http://www.symfony-project.org/book/1_1/08-Inside-the-Model-Layer

```
Author:
  columns:
    first_name: string(20)
    last_name: string(20)
    email: string(255)
    active: boolean
Category:
  columns:
    name: string(255)
Tag:
  columns:
    name: string(255)
ArticleTag:
  columns:
    article_id:
      type: integer
      primary: true
    tag_id:
      type: integer
      primary: true
  relations:
    Article:
      onDelete: CASCADE
    Tag:
      onDelete: CASCADE
```

Here are the relations between the tables:

- 1-n relation between the `article` table and the `author` table: an article is written by one and only one author
- 1-n relation between the `article` table and the `category` table: an article belongs to one or zero category
- n-n relation between the `article` and `tag` tables

Generating Form Classes

We want to edit the information of the `article`, `author`, `category`, and `tag` tables. To do so, we need to create forms linked to each of these tables and configure widgets and validators related to the database schema. Even if it is possible to create these forms manually, it is a long, tedious task, and overall, it forces repetition of the same kind of information in several files (column and field name, maximum size of column and fields, ...). Furthermore, each time we change the model, we will also have to change the related form class. Fortunately, the Doctrine plugin has a built-in task `doctrine:build-forms` that automates this process generating the forms related to the object model:

Listing 6-2
```
$ ./symfony doctrine:build-forms
```

During the form generation, the task creates one class per table with validators and widgets for each column using introspection of the model and taking into account relations between tables.

The `doctrine:build-all` and `doctrine:build-all-load` also updates form classes, automatically invoking the `doctrine:build-forms` task.

After executing these tasks, a file structure is created in the `lib/form/` directory. Here are the files created for our example schema:

Listing 6-3

```
lib/
  form/
    doctrine/
      ArticleForm.class.php
      ArticleTagForm.class.php
      AuthorForm.class.php
      CategoryForm.class.php
      TagForm.class.php
      base/
        BaseArticleForm.class.php
        BaseArticleTagForm.class.php
        BaseAuthorForm.class.php
        BaseCategoryForm.class.php
        BaseFormDoctrine.class.php
        BaseTagForm.class.php
```

The `doctrine:build-forms` task generates two classes for each table of the schema, one base class in the `lib/form/base` directory and one in the `lib/form/` directory. For example, the `author` table, consists of `BaseAuthorForm` and `AuthorForm` classes that were generated in the files `lib/form/base/BaseAuthorForm.class.php` and `lib/form/AuthorForm.class.php`.

Table below sums up the hierarchy among the different classes involved in the `AuthorForm` form definition.

Class	Package	For	Description
AuthorForm	project	developer	Overrides generated form
BaseAuthorForm	project	symfony	Based on the schema and overridden at each execution of the `doctrine:build-forms` task
BaseFormDoctrine	project	developer	Allows global Customization of Doctrine forms
sfFormDoctrine	Doctrine plugin	symfony	Base of Doctrine forms
sfForm	symfony	symfony	Base of symfony forms

In order to create or edit an object from the `Author` class, we will use the `AuthorForm` class, described in Listing 4-2. As you can notice, this class does not contain any methods as it inherits from the `BaseAuthorForm` which is generated through the configuration. The `AuthorForm` class is the class we will use to Customize and override the form configuration.

Listing 4-2 - `AuthorForm` *Class*

Listing 6-4

```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
  }
}
```

Listing 4-3 shows the `BaseAuthorForm` class with the validators and widgets generated introspecting the model for the `author` table.

Listing 4-3 - `BaseAuthorForm` *Class representing the Form for the* `author` *table*

Listing 6-5

```
class BaseAuthorForm extends BaseFormDoctrine
{
```

```
  public function setup()
  {
    $this->setWidgets(array(
      'id'         => new sfWidgetFormInputHidden(),
      'first_name' => new sfWidgetFormInput(),
      'last_name'  => new sfWidgetFormInput(),
      'email'      => new sfWidgetFormInput(),
    ));

    $this->setValidators(array(
      'id'         => new sfValidatorDoctrineChoice(array('model' =>
'Author', 'column' => 'id', 'required' => false)),
      'first_name' => new sfValidatorString(array('max_length' => 20,
'required' => false)),
      'last_name'  => new sfValidatorString(array('max_length' => 20,
'required' => false)),
      'email'      => new sfValidatorString(array('max_length' => 255)),
    ));

    $this->widgetSchema->setNameFormat('author[%s]');

    $this->errorSchema = new
sfValidatorErrorSchema($this->validatorSchema);

    parent::setup();
  }

  public function getModelName()
  {
    return 'Author';
  }
}
```

The generated class looks very similar to the forms we have already created in the previous chapters, except for a few things:

- The base class is `BaseFormDoctrine` instead of `sfForm`
- The validator and widget configuration takes place in the `setup()` method, rather than in the `configure()` method
- The `getModelName()` method returns the Doctrine class related to this form

Global Customization of Doctrine Forms

In addition to the classes generated for each table, the `doctrine:build-forms` also generates a `BaseFormDoctrine` class. This empty class is the base class of every other generated class in the `lib/form/base/` directory and allows to configure the behavior of every Doctrine form globally. For example, it is possible to easily change the default formatter for all Doctrine forms:

Listing 6-6

```
abstract class BaseFormDoctrine extends sfFormDoctrine
{
  public function setup()
  {
    sfWidgetFormSchema::setDefaultFormFormatterName('div');
  }
}
```

You'll notice that the `BaseFormDoctrine` class inherits from the `sfFormDoctrine` class. This class incorporates functionality specific to Doctrine and among other things deals with the object serialization in database from the values submitted in the form.

TIP Base classes use the `setup()` method for the configuration instead of the `configure()` method. This allows the developer to override the configuration of empty generated classes without handling the `parent::configure()` call.

The form field names are identical to the column names we set in the schema: `id`, `first_name`, `last_name`, and `email`.

For each column of the `author` table, the `doctrine:build-forms` task generates a widget and a validator according to the schema definition. The task always generates the most secure validators possible. Let's consider the `id` field. We could just check if the value is a valid integer. Instead the validator generated here allows us to also validate that the identifier actually exists (to edit an existing object) or that the identifier is empty (so that we could create a new object). This is a stronger validation.

The generated forms can be used immediately. Add a `<?php echo $form ?>` statement, and this will allow to create functional forms with validation **without writing a single line of code**.

Beyond the ability to quickly make prototypes, generated forms are easy to extend without having to modify the generated classes. This is thanks to the inheritance mechanism of the base and form classes.

At last at each evolution of the database schema, the task allows to generate again the forms to take into account the schema modifications, without overriding the Customization you might have made.

The CRUD Generator

Now that there are generated form classes, let's see how easy it is to create a symfony module to deal with the objects from a browser. We wish to create, modify, and delete objects from the `Article`, `Author`, `Category`, and `Tag` classes. Let's start with the module creation for the `Author` class. Even if we can manually create a module, the Doctrine plugin provides the `doctrine:generate-crud` task which generates a CRUD module based on a Doctrine object model class. Using the form we generated in the previous section:

Listing 6-7

```
$ ./symfony doctrine:generate-crud frontend author Author
```

The `doctrine:generate-crud` takes three arguments:

- `frontend` : name of the application you want to create the module in
- `author` : name of the module you want to create
- `Author` : name of the model class you want to create the module for

CRUD stands for Creation / Retrieval / Update / Deletion and sums up the four basic operations we can carry out with the model datas.

In Listing 4-4, we see that the task generated five actions allowing us to list (`index`), create (`create`), modify (`edit`), save (`update`), and delete (`delete`) the objects of the `Author` class.

Listing 4-4 - The `authorActions` *Class generated by the Task*

Listing 6-8

```
// apps/frontend/modules/author/actions/actions.class.php
class authorActions extends sfActions
{
  public function executeIndex()
  {
    $this->authorList = $this->getAuthorTable()->findAll();
  }

  public function executeCreate()
  {
    $this->form = new AuthorForm();

    $this->setTemplate('edit');
  }

  public function executeEdit($request)
  {
    $this->form = $this->getAuthorForm($request->getParameter('id'));
  }

  public function executeUpdate($request)
  {
    $this->forward404Unless($request->isMethod('post'));

    $this->form = $this->getAuthorForm($request->getParameter('id'));

    $this->form->bind($request->getParameter('author'));
    if ($this->form->isValid())
    {
      $author = $this->form->save();

      $this->redirect('author/edit?id='.$author->get('id'));
    }

    $this->setTemplate('edit');
  }

  public function executeDelete($request)
  {
    $this->forward404Unless($author =
$this->getAuthorById($request->getParameter('id')));

    $author->delete();
```

```
    $this->redirect('author/index');
  }

  private function getAuthorTable()
  {
    return Doctrine::getTable('Author');
  }

  private function getAuthorById($id)
  {
    return $this->getAuthorTable()->find($id);
  }

  private function getAuthorForm($id)
  {
    $author = $this->getAuthorById($id);

    if ($author instanceof Author)
    {
      return new ArticleForm($author);
    }
    else
    {
      return new ArticleForm();
    }
  }
}
```

In this module, the form life cycle is handled by three methods: `create`, `edit` and, `update`. It is also possible to ask the `doctrine:generate-crud` task to generate only one method covering the three previous methods functionalities, with the option `--non-atomic-actions`:

```
$ ./symfony doctrine:generate-crud frontend author Author
--non-atomic-actions
```

Listing 6-9

The generated code using `--non-atomic-actions` (Listing 4-5) is more concise and less verbose.

Listing 4-5 - The `authorActions` *Class generated with the* `--non-atomic-actions` *option*

```
class authorActions extends sfActions
{
  public function executeIndex()
  {
    $this->authorList = $this->getAuthorTable()->findAll();
  }

  public function executeEdit($request)
  {
    $this->form = new
AuthorForm(Doctrine::getTable('Author')->find($request->getParameter('id')));

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('author'));
      if ($this->form->isValid())
      {
        $author = $this->form->save();
```

Listing 6-10

```
        $this->redirect('author/edit?id='.$author->getId());
      }
    }
  }

  public function executeDelete($request)
  {
    $this->forward404Unless($author =
Doctrine::getTable('Author')->find($request->getParameter('id')));

    $author->delete();

    $this->redirect('author/index');
  }
}
```

The task also generated two templates, `indexSuccess` and `editSuccess`. The `editSuccess` template was generated without using the `<?php echo $form ?>` statement. We can modify this behavior, using the `--non-verbose-templates`:

Listing 6-11

```
$ ./symfony doctrine:generate-crud frontend author Author
--non-verbose-templates
```

This option is helpful during prototyping phases, as Listing 4-6 shows.

Listing 4-6 - The `editSuccess` *Template*

Listing 6-12

```
// apps/frontend/modules/author/templates/editSuccess.php
<?php $author = $form->getObject() ?>
<h1><?php echo $author->isNew() ? 'New' : 'Edit' ?> Author</h1>

<form action="<?php echo url_for('author/edit'.(!$author->isNew() ?
'?id='.$author->getId() : '')) ?>" method="post" <?php
$form->isMultipart() and print 'enctype="multipart/form-data" ' ?>>
  <table>
    <tfoot>
      <tr>
        <td colspan="2">
           <a href="<?php echo url_for('author/index') ?>">Cancel</a>
          <?php if (!$author->isNew()): ?>
             <?php echo link_to('Delete', 'author/
delete?id='.$author->getId(), array('post' => true, 'confirm' => 'Are you
sure?')) ?>
          <?php endif; ?>
          <input type="submit" value="Save" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

The `--with-show` option let us generate an action and a template we can use to view an object (read only).

You can now open the URL `/frontend_dev.php/author` in a browser to view the generated module (Figure 4-1 and Figure 4-2). Take time to play with the interface. Thanks to the generated module you can list the authors, add a new one, edit, modify, and even delete. You will also notice that the validation rules are also working.

Figure 4-1 - Authors List

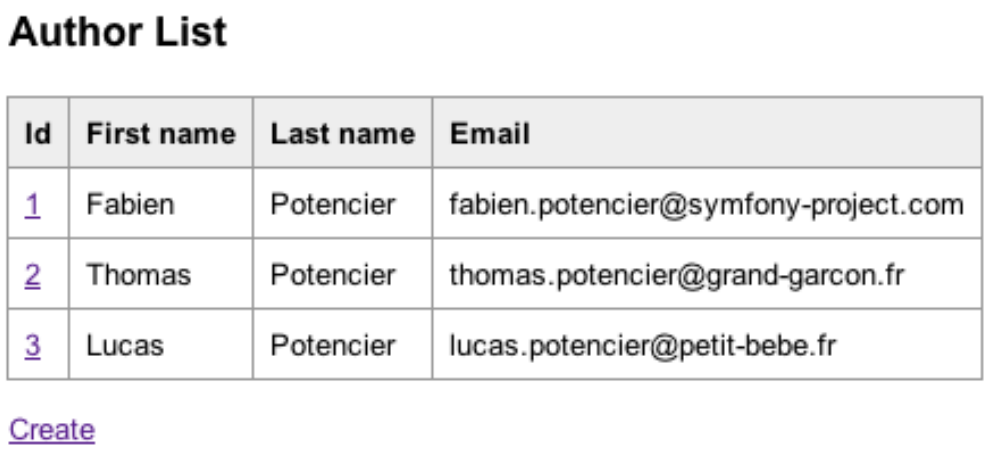
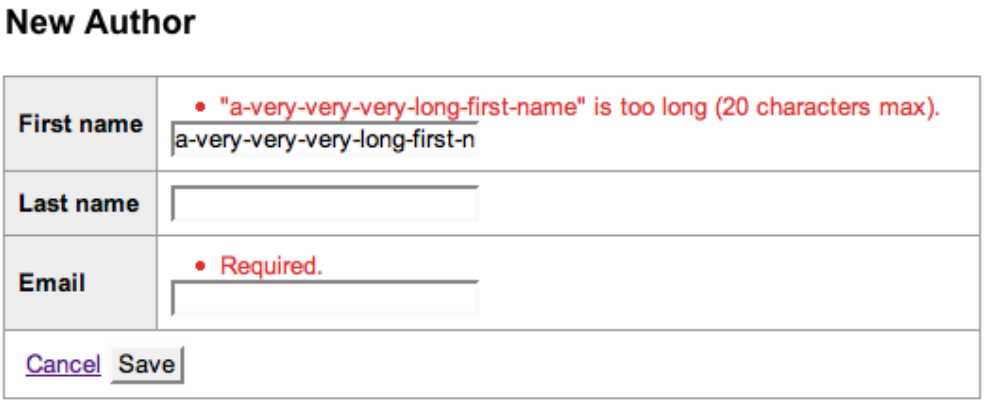


Figure 4-2 - Editing an Author with Validation Errors



We can now repeat the operation with the `Article` class:

```
$ ./symfony doctrine:generate-crud frontend article Article
--non-verbose-templates --non-atomic-actions
```

Listing 6-13

The generated code is quite similar to the code of the `Author` class. However, if you try to create a new article, the code throws a fatal error as you can see in Figure 4-3.

Figure 4-3 - Linked Tables must define the `__toString()` method

The `ArticleForm` form uses the `sfWidgetFormDoctrineSelect` widget to represent the relation between the `Article` object and the `Author` object. This widget creates a drop-down list with the authors. During the display, the authors objects are converted into a string of characters using the `__toString()` magic method, which must be defined in the `Author` class as shown in Listing 4-7.

Listing 4-7 - Implementing the `__toString()` method for the `Author` class

Listing 6-14
```
class Author extends BaseAuthor
{
  public function __toString()
  {
    return $this->getFirstName().' '.$this->getLastName();
  }
}
```

Just like the `Author` class, you can create `__toString()` methods for the other classes of our model: `Article`, `Category`, and `Tag`.

sfDoctrineRecord will attempt to guess in the base __toString() if you do not specify your own. It checks for columns named title, name, subject, etc. to use as the string representation.

Tip The `method` option of the `sfWidgetFormDoctrineSelect` widget change the method used to represent an object in text format.

The Figure 4-4 Shows how to create an article after having implemented the `__toString()` method.

Figure 4-4 - Creating an Article

Customizing the generated Forms

The `doctrine:build-forms` and `doctrine:generate-crud` tasks let us create functional symfony modules to list, create, edit, and delete model objects. These modules are taking into account not only the validation rules of the model but also the relationships between tables. All of this happens without writing a single line of code!

The time has now come to customize the generated code. If the form classes are already considering many elements, some aspects will need to be customized.

Configuring validators and widgets

Let's start with configuring the validators and widgets generated by default.

The `ArticleForm` form has a `slug` field. The slug is a string of characters that uniquely representing the article in the URL. For instance, the slug of an article whose title is "Optimize the developments with symfony" is `12-optimize-the-developments-with-symfony`, `12` being the article `id`. This field is usually automatically computed when the object is saved, depending on the `title`, but it has the potential to be explicitly overridden by the user. Even if this field is required in the schema, it can not be compulsory to the form. That is why we

modify the validator and make it optional, as in Listing 4-8. We will also customize the `content` field increasing its size and forcing the user to type in at least five characters.

Listing 4-8 - Customizing Validators and Widgets

Listing 6-15
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $this->validatorSchema['slug']->setOption('required', false);
    $this->validatorSchema['content']->setOption('min_length', 5);

    $this->widgetSchema['content']->setAttributes(array('rows' => 10,
'cols' => 40));
  }
}
```

We use here the `validatorSchema` and `widgetSchema` objects as PHP arrays. These arrays are taking the name of a field as key and return respectively the validator object and the related widget object. We can then Customize individually fields and widgets.

In order to allow the use of objects as PHP arrays, the `sfValidatorSchema` and `sfWidgetFormSchema` classes implement the `ArrayAccess` interface, available in PHP since version 5.

To make sure two articles can not have the same `slug`, a uniqueness constraint has been added in the schema definition. This constraint on the database level is reflected in the `ArticleForm` form using the `sfValidatorDoctrineUnique` validator. This validator can check the uniqueness of any form field. It is helpful among other things to check the uniqueness of an email address of a login for instance. Listing 4-9 shows how to use it in the `ArticleForm` form.

Listing 4-9 - Using the `sfValidatorDoctrineUnique` validator to check the Uniqueness of a field

Listing 6-16
```
class BaseArticleForm extends BaseFormDoctrine
{
  public function setup()
  {
    // ...

    $this->validatorSchema->setPostValidator(
      new sfValidatorDoctrineUnique(array('model' => 'Article', 'column'
=> array('slug')))
    );
  }
}
```

The `sfValidatorDoctrineUnique` validator is a `postValidator` running on the whole data after the individual validation of each field. In order to validate the `slug` uniqueness, the validator must be able to access, not only the `slug` value, but also the value of the primary key(s). Validation rules are indeed different throughout the creation and the edition since the slug can stay the same during the update of an article.

Let's Customize now the `active` field of the `author` table, used to know if an author is active. Listing 4-10 shows how to exclude inactive authors from the `ArticleForm` form, modifying the

`query` option of the `sfWidgetDoctrineSelect` widget connected to the `author_id` field. The `query` option accepts a Doctrine Query object, allowing to narrow down the list of available options in the rolling list.

Listing 4-10 - Customizing the `sfWidgetDoctrineSelect` *widget*

Listing 6-17

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $query = Doctrine_Query::create()
      ->from('Author a')
      ->where('a.active = ?', true);
    $this->widgetSchema['author_id']->setOption('query', $query);
  }
}
```

Even if the widget customization can make us narrow down the list of available options, we must not forget to consider this narrowing on the validator level, as shown in Listing 4-11. Like the `sfWidgetProperSelect` widget, the `sfValidatorDoctrineChoice` validator accepts a `query` option to narrow down the options valid for a field.

Listing 4-11 - Customizing the `sfValidatorDoctrineChoice` *validator*

Listing 6-18

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $query = Doctrine_Query::create()
      ->from('Author a')
      ->where('a.active = ?', true);

    $this->widgetSchema['author_id']->setOption('query', $query);
    $this->validatorSchema['author_id']->setOption('query', $query);
  }
}
```

In the previous example we defined the `Query` object directly in the `configure()` method. In our project, this query will certainly be helpful in other circumstances, so it is better to create a `getActiveAuthorsQuery()` method within the `AuthorTable` class and to call this method from `ArticleForm` as Listing 4-12 shows.

Listing 4-12 - Refactoring the `Query` *in the Model*

Listing 6-19

```
class AuthorTable extends Doctrine_Table
{
  public function getActiveAuthorsQuery()
  {
    $query = Doctrine_Query::create()
      ->from('Author a')
      ->where('a.active = ?', true);

    return $query;
  }
}
```

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $authorQuery = Doctrine::getTable('Author')->getActiveAuthorsQuery();
    $this->widgetSchema['author_id']->setOption('query', $authorQuery);
    $this->validatorSchema['author_id']->setOption('query', $authorQuery);
  }
}
```

Like the `sfWidgetDoctrineSelect` widget and the `sfValidatorDoctrineChoice` validator represent a 1-n relation between two tables, the `sfWidgetDoctrineSelectMany` and the `sfValidatorDoctrineChoiceMany` validator represent a n-n relation and accept the same options. In the `ArticleForm` form, these classes are used to represent a relation between the `article` table and the `tag` table.

Changing validator

The `email` being defined as a `string(255)` in the schema, symfony created a `sfValidatorString()` validator restraining the maximum length to 255 characters. This field is also supposed to receive a valid email, Listing 4-14 replaces the generated validator with a `sfValidatorEmail` validator.

Listing 4-13 - Changing the `email` *field Validator of the* `AuthorForm` *class*

Listing 6-20
```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorEmail();
  }
}
```

Adding a validator

We observed in the previous chapter how to modify the generated validator. But in the case of the `email` field, it would be useful to keep the maximum length validation. In Listing 4-14, we use the `sfValidatorAnd` validator to guarantee the email validity and check the maximum length allowed for the field.

Listing 4-14 - Using a multiple Validator

Listing 6-21
```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      new sfValidatorString(array('max_length' => 255)),
      new sfValidatorEmail(),
    ));
  }
}
```

The previous example is not perfect, because if we decide later to modify the length of the `email` field in the database schema, we will have to think about doing it also in the form. Instead of replacing the generated validator, it is better to add one, as shown in Listing 4-15.

Listing 4-15 - Adding a Validator

Listing 6-22

```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));
  }
}
```

Changing widget

In the database schema, the `status` field of the `article` table stores the article status as a string of characters. The possible values were defined in the `ArticeTable` class, as shown in Listing 4-16.

Listing 4-16 - Defining available Statuses in the `ArticleTable` class

Listing 6-23

```
class ArticleTable extends Doctrine_Table
{
  static protected $statuses = array('draft', 'online', 'offline');

  static public function getStatuses()
  {
    return self::$statuses;
  }

  // ...
}
```

When editing an article, the `status` field must be represented as a drop-down list instead of a text field. To do so, let's change the widget we used, as shown in Listing 4-17.

Listing 4-17 - Changing the Widget for the `status` field

Listing 6-24

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
=> ArticleTable::getStatuses()));
  }
}
```

To be thorough we must also change the validator to make sure the chosen status actually belongs to the list of possible options (Listing 4-18).

Listing 4-18 - Modifying the `status` Field Validator

Listing 6-25

```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $statuses = ArticleTable::getStatuses();

    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
```

```
=> $statuses));

    $this->validatorSchema['status'] = new
sfValidatorChoice(array('choices' => array_keys($statuses)));
  }
}
```

Deleting a field

The `article` table has two special columns, `created_at` and `updated_at`, whose update is automatically handled by Doctrine. We must then delete them from the form as Listing 4-19 show, to prevent the user from modifying them.

Listing 4-19 - Deleting a Field

Listing 6-26
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    unset($this->validatorSchema['created_at']);
    unset($this->widgetSchema['created_at']);

    unset($this->validatorSchema['updated_at']);
    unset($this->widgetSchema['updated_at']);
  }
}
```

In order to delete a field, it is necessary to delete its validator and its widget. Listing 4-20 shows how it is also possible to delete both in one action, using the form as a PHP array.

Listing 4-20 - Deleting a Field using the Form as a PHP Array

Listing 6-27
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    unset($this['created_at'], $this['updated_at']);
  }
}
```

Sum up

To sum up, Listing 4-21 and Listing 4-22 show the `ArticleForm` and `AuthorForm` forms as we customize them.

Listing 4-21 - `ArticleForm` *Form*

Listing 6-28
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    $authorQuery = Doctrine::getTable('Author')->getActiveAuthorsQuery();

    // widgets
    $this->widgetSchema['content']->setAttributes(array('rows' => 10,
'cols' => 40));
    $this->widgetSchema['status'] = new sfWidgetFormSelect(array('choices'
=> ArticleTable::getStatuses()));
    $this->widgetSchema['author_id']->setOption('query', $authorQuery);
```

```
    // validators
    $this->validatorSchema['slug']->setOption('required', false);
    $this->validatorSchema['content']->setOption('min_length', 5);
    $this->validatorSchema['status'] = new
sfValidatorChoice(array('choices' =>
array_keys(ArticleTable::getStatuses())));
    $this->validatorSchema['author_id']->setOption('query', $authorQuery);

    unset($this['created_at']);
    unset($this['updated_at']);
  }
}
```

Listing 4-22 - `AuthorForm` *Form*

Listing 6-29

```
class AuthorForm extends BaseAuthorForm
{
  public function configure()
  {
    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));
  }
}
```

Using the `doctrine:build-forms` allows to automatically generate most of the elements letting forms introspect the object model. This automatization is helpful for several reasons:

- It makes the developer's life easier, saving him from a repetitive and redundant work. He can then focus on the validators and widget Customization according to the project's specific business rules .
- Besides, when the database schema is updated, the generated forms will be automatically updated. The developer will just have to tune the customization they made.

The next section will describe the customization of actions and templates generated by the `doctrine:generate-crud` task.

Form Serialization

The previous section show us how to customize forms generated by the task `doctrine:build-forms`. In the current section, we will customize the life cycle of forms, starting from the code generated by the `doctrine:generate-crud` task.

Default values

A Doctrine form instance is always connected to a Doctrine object. The linked Doctrine object always belongs to the class returned by the `getModelName()` method. For instance, the `AuthorForm` form can only be linked to objects belonging to the `Author` class. This object is either an empty object (a blank instance of the `Author` class), or the object sent to the constructor as first argument. Whereas the constructor of an "average" form takes an array of values as first argument, the constructor of a Doctrine form takes a Doctrine object. This object is used to define each form field default value. The `getObject()` method returns the

object related to the current instance and the `isNew()` method allows to know if the object was sent via the constructor:

Listing 6-30
```
// creating a new object
$authorForm = new AuthorForm();

print $authorForm->getObject()->getId(); // outputs null
print $authorForm->isNew();              // outputs true

// modifying an existing object
$author = Doctrine::getTable('Author')->find(1);
$authorForm = new AuthorForm($author);

print $authorForm->getObject()->getId(); // outputs 1
print $authorForm->isNew();              // outputs false
```

Handling life cycle

As we observed at the beginning of the chapter, the `edit` action, shown in Listing 4-23, handles the form life cycle.

Listing 4-23 - The `executeEdit` *Method of the* `author` *Module*

Listing 6-31
```
// apps/frontend/modules/author/actions/actions.class.php
class authorActions extends sfActions
{
  // ...

  public function executeEdit($request)
  {
    $author =
Doctrine::getTable('Author')->find($request->getParameter('id'));
    $this->form = new AuthorForm($author);

    if ($request->isMethod('post'))
    {
      $this->form->bind($request->getParameter('author'));
      if ($this->form->isValid())
      {
        $author = $this->form->save();

        $this->redirect('author/edit?id='.$author->getId());
      }
    }
  }
}
```

Even if the `edit` action looks like the actions we might have describe in the previous chapters, we can point a few differences:

- A Doctrine object from the `Author` class is sent as first argument to the form constructor:

Listing 6-32
```
$author =
Doctrine::getTable('Author')->find($request->getParameter('id'));
$this->form = new AuthorForm($author);
```

- The widgets `name` attribute format is automatically customize to allow the retrieval of the input data in a PHP array named after the related table (`author`):

```
$this->form->bind($request->getParameter('author'));
```

Listing 6-33

- When the form is valid, a mere call to the `save()` method creates or updates the Doctrine object related to the form:

```
$author = $this->form->save();
```

Listing 6-34

Creating and Modifying a Doctrine Object

Listing 4-23 code handles with a single method the creation and modification of objects from the `Author` class:

- Creation of a new `Author` object:
  - The `index` action is called with no `id` parameter (`$request->getParameter('id')` is `null`)
  - The call to the `find()` therefore sends `null`
  - The `form` object is then linked to an empty `Author` Doctrine object
  - The `$this->form->save()` call creates consequently a new `Author` object when a valid form is submitted
- Modification of an existing `Author` object:
  - The `index` action is called with an `id` parameter (`$request->getParameter('id')` standing for the primary key the `Author` object is to modify)
  - The call to the `find()` method returns the `Author` object related to the primary key
  - The `form` object is therefore linked to the previously found object
  - The `$this->form->save()` call updates the `Author` object when a valid form is submitted

The `save()` method

When a Doctrine form is valid, the `save()` method updates the related object and stores it in the database. This method actually stores not only the main object but also the potentially related objects. For instance, the `ArticleForm` form updates the tags connected to an article. The relation between the `article` table and the `tag` table being a n-n relation, the tags related to an article are saved in the `article_tag` table (using the `saveArticleTagList()` generated method).

In order to certify a consistent serialization, the `save()` method includes every updates in one transaction.

We will see in Chapter 9 that the `save()` method also automatically updates the internationalized tables.

SIDEBAR Using the `bindAndSave()` method

The `bindAndSave()` method binds the input data the user submitted to the form, validates this form and updates the related object in the database, all in one operation:

Listing 6-35
```
class articleActions extends sfActions
{
  public function executeCreate(sfWebRequest $request)
  {
    $this->form = new ArticleForm();

    if ($request->isMethod('post') &&
$this->form->bindAndSave($request->getParameter('article')))
    {
      $this->redirect('article/created');
    }
  }
}
```

Handling the files upload

The `save()` method automatically updates the Doctrine objects but can not handle the side elements as managing the file upload.

Let's see how to attach a file to each article. Files are stored in the `web/uploads` directory and a reference to the file path is kept in the `file` field of the `article` table, as shown in Listing 4-24.

Listing 4-24 - Schema for the `article` *Table with associated File*

Listing 6-36
```
// config/schema.yml
doctrine:
  article:
    // ...
    file: string(255)
```

After every schema update, you need to update the object model, the database and the related forms:

Listing 6-37
```
$ ./symfony doctrine:build-all
```

Do mind that the `doctrine:build-all` task deletes every schema tables to re-create them. The data inside the tables are therefore overwritten. That is why it is important to create test data (`fixtures`) you can download again at each model modification.

Listing 4-25 shows how to modify the `ArticleForm` class in order to link a widget and a validator to the `file` field.

Listing 4-25 - Modifying the `file` *Field of the* `ArticleForm` *form.*

Listing 6-38
```
class ArticleForm extends BaseArticleForm
{
  public function configure()
  {
    // ...

    $this->widgetSchema['file'] = new sfWidgetFormInputFile();
    $this->validatorSchema['file'] = new sfValidatorFile();
  }
}
```

As for every form allowing to upload a file, does not forget to add also the `enctype` attribute to the `form` tag of the template (see Chapter 2 for further informations concerning file upload management).

Listing 4-26 shows the modifications to apply when saving the form to upload the file onto the server and store its path in the `article` object.

Listing 4-26 - Saving the `article` *Object and the File uploaded in the Action*

Listing 6-39

```
public function executeEdit($request)
{
  $author =
Doctrine::getTable('Author')->find($request->getParameter('id'));
  $this->form = new ArticleForm($author);

  if ($request->isMethod('post'))
  {
    $this->form->bind($request->getParameter('article'),
$request->getFiles('article'));
    if ($this->form->isValid())
    {
      $file = $this->form->getValue('file');
      $filename =
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
      $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

      $article = $this->form->save();

      $this->redirect('article/edit?id='.$article->getId());
    }
  }
}
```

Saving the uploaded file on the filesystem allows the `sfValidatedFile` object to know the absolute path to the file. During the call to the `save()` method, the fields values are used to update the related object and, as for the `file` field, the `sfValidatedFile` object is converted in a character string thanks to the `__toString()` method, sending back the absolute path to the file. The `file` column of the `article` table will store this absolute path.

If you wish to store the path relative to the `sfConfig::get('sf_upload_dir')` directory, you can create a class inheriting from `sfValidatedFile` and use the `validated_file_class` option to send to the `sfValidatorFile` validator the name of the new class. The validator will then return an instance of your class. We will see in the rest of this chapter another approach, consisting in modifying the value of the `file` column before saving the object in database.

Customizing the `save()` method

We observed in the previous section how to save the uploaded file in the `edit` action. One of the principles of the object oriented programming is the reusability of the code, thanks to its encapsulation in classes. Instead of duplicating the code used to save the file in each action using the `ArticleForm` form, it is better to move it in the `ArticleForm` class. Listing 4-27 shows how to override the `save()` method in order to also save the file and possibly to delete of an existing file.

Listing 4-27 - Overriding the `save()` *Method of the* `ArticleForm` *Class*

Listing 6-40
```
class ArticleForm extends BaseFormDoctrine
{
  // ...

  public function save($con = null)
  {
    if (file_exists($this->getObject()->getFile()))
    {
      unlink($this->getObject()->getFile());
    }

    $file = $this->getValue('file');
    $filename =
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
    $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

    return parent::save($con);
  }
}
```

After moving the code to the form, the `edit` action is identical to the code initially generated by the `doctrine:generate-crud` task.

Refactoring the Code in the Model of in the Form

The actions generated by the `doctrine:generate-crud` task shouldn't usually be modified.

The logic you could add in the `edit` action, especially during the form serialization, must usually be moved in the model classes or in the form class.

We just went over an example of refactoring in the form class in order to consider a uploaded file storing. Let's take another example related to the model. The `ArticleForm` form has a `slug` field. We observed that this field should be automatically computed from the `title` field name that it should be potentially overridden by the user. This logic does not depend on the form. It belongs therefore to the model, as shown the following code:

Listing 6-41
```
class Article extends BaseArticle
{
  public function save($con = null)
  {
    if (!$this->getSlug())
    {
      $this->setSlugFromTitle();
    }

    return parent::save($con);
  }

  protection function setSlugFromTitle()
  {
    // ...
  }
}
```

The main goal of those refactorings is to respect the separation in applicative layers, and especially the reusability of the developments.

Customizing the `doSave()` method

We observed that the saving of an object was made within a transaction in order to guarantee that each operation related to the saving is processed correctly. When overriding the `save()`method as we did in the previous section in order to save the uploaded file, the executed code is independent from this transaction.

Listing 4-28 shows how to use the `doSave()` method to insert in the global transaction our code saving the uploaded file.

Listing 4-28 - Overriding the `doSave()` *Method in the* `ArticleForm` *Form*

Listing 6-42

```
class ArticleForm extends BaseFormDoctrine
{
  // ...

  public function doSave($con = null)
  {
    if (file_exists($this->getObject()->getFile()))
    {
      unlink($this->getObject()->getFile());
    }

    $file = $this->getValue('file');
    $filename =
sha1($file->getOriginalName()).$file->getExtension($file->getOriginalExtension());
    $file->save(sfConfig::get('sf_upload_dir').'/'.$filename);

    return parent::doSave($con);
  }
}
```

The `doSave()` method being called in the transaction created by the `save()` method, if the call to the `save()` method of the `file()` object throws an exception, the object will not be saved.

Customizing the `updateObject()` Method

It is sometimes necessary to modify the object connected to the form between the update and the saving in database.

In our file upload example, instead of storing the absolute path to the uploaded file in the `file` column, we wish to store the path relative to the `sfConfig::get('sf_upload_dir')` directory.

Listing 4-29 shows how to override the `updateObject()` method of the `ArticleForm` form in order to change the value of the `file` column after the automatic update object but before it is saved.

Listing 4-29 - Overriding the `updateObject()` *Method and the* `ArticleForm` *Class*

Listing 6-43

```
class ArticleForm extends BaseFormDoctrine
{
  // ...

  public function updateObject($values = null)
  {
    $object = parent::updateObject($values);

    $object->setFile(str_replace(sfConfig::get('sf_upload_dir').'/', '',
$object->getFile()));
```

```
    return $object;
  }
}
```

The `updateObject()` method is called by the `doSave()` method before saving the object in database.