PART 4

Development Tools

CHAPTER 14

Generators

Many applications are based on data stored in a database and offer an interface to access it. Symfony automates the repetitive task of creating a module providing data manipulation capabilities based on a Propel object. If your object model is properly defined, symfony can even generate an entire site administration automatically. This chapter will tell you of the two generators bundled in symfony: scaffolding and administration generator. The latter relies on a special configuration file with a complete syntax, so most of this chapter describes the various possibilities of the administration generator.

Code Generation Based on the Model

In a web application, data access operations can be categorized as one of the following:

- Creation of a record
- Retrieval of records
- Update of a record (and modification of its columns)
- Deletion of a record

These operations are so common that they have a dedicated acronym: *CRUD*. Many pages can be reduced to one of them. For instance, in a forum application, the list of latest posts is a retrieve operation, and the reply to a post corresponds to a create operation.

The basic actions and templates that implement the CRUD operations for a given table are repeatedly created in web applications. In symfony, the model layer contains enough information to allow generating the CRUD operations code, so as to speed up the early part of the development or the back-end interfaces.

All the code generation tasks based on the model create *an entire module*, and result from a single call to the symfony command line in the shape of the following:

```
> symfony <TASK_NAME> <APP_NAME> <MODULE_NAME> <CLASS_NAME>
```

The code generation tasks are `propel-init-crud`, `propel-generate-crud`, and `propel-init-admin`.

Scaffolding and Administration

During application development, code generation can be used for two distinct purposes:

> A *scaffolding* is the basic structure (actions and templates) required to operate CRUD on a given table. The code is minimal, since it is meant to serve as a guideline for further development. It is a starting base that must be adapted to match your logic and presentation requirements. Scaffoldings are mostly used during the development phase, to provide a web access to a database, to build a prototype, or to bootstrap a module primarily based on a table.

> An *administration* is a sophisticated interface for data manipulation, dedicated to backend administration. Administrations differ from scaffoldings because their code is not meant to be modified manually. They can be customized, extended, or assembled through configuration or inheritance. Their presentation is important, and they take advantage of additional features such as sorting, pagination, and filtering. An administration can be created and handed over to the client as a finished part of the software product.

The symfony command line uses the word *crud* to designate a scaffolding, and *admin* for an administration.

Initiating or Generating Code

Symfony offers two ways to generate code: either by inheritance (`init`) or by code generation (`generate`).

You can *initiate* a module, that is, create empty classes that inherit from the framework. This masks the PHP code of the actions and the templates to avoid them from being modified. This is useful if your data structure is not final, or if you just need a quick interface to a database to manipulate records. The code executed at runtime is not located in your application, but in the cache. The command-line tasks for this kind of generation start with `propel-init-`.

Initiated action code is empty. For instance, an initiated `article` module has actions looking like this:

```
class articleActions extends autoarticleActions
{
}
```

On the other hand, you can also *generate* the code of the actions and the templates so that it can be modified. The resulting module is therefore independent from the classes of the framework, and cannot be altered using configuration files. The command-line tasks for this kind of generation start with `propel-generate-`.

As the scaffoldings are built to serve as a base for further developments, it is often best to *generate* a scaffolding. On the other hand, an administration should be easy to update through a change in the configuration, and it should remain usable even if the data model changes. That's why administrations are *initiated* only.

Example Data Model

Throughout this chapter, the listings will demonstrate the capabilities of the symfony generators based on a simple example, which will remind you of Chapter 8. This is the well-known example of

the weblog application, containing two `Article` and `Comment` classes. Listing 14-1 shows its schema, illustrated in Figure 14-1.

Listing 14-1. *`schema.yml` of the Example Weblog Application*

```
propel:
  blog_article:
    _attributes: { phpName: Article }
    id:
    title:       varchar(255)
    content:     longvarchar
    created_at:
  blog_comment:
    _attributes: { phpName: Comment }
    id:
    article_id:
    author:      varchar(255)
    content:     longvarchar
    created_at:
```

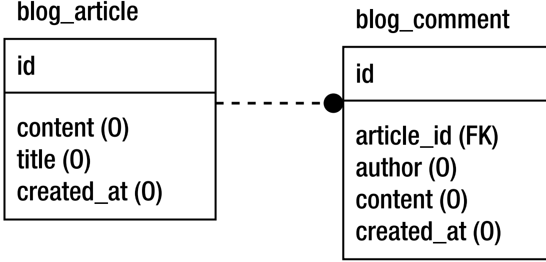


Figure 14-1. *Example data model*

There is no particular rule to follow during the schema creation to allow code generation. Symfony will use the schema as is and interpret its attributes to generate a scaffolding or an administration.

■**Tip** To get the most out of this chapter, you need to actually do the examples. You will get a better understanding of what symfony generates and what can be done with the generated code if you have a view of every step described in the listings. So you are invited to create a data structure such as the one described previously, to create a database with a `blog_article` and a `blog_comment` table, and to populate this database with sample data.

Scaffolding

Scaffolding is of great use in the early days of application development. With a single command, symfony creates an entire module based on the description of a given table.

Generating a Scaffolding

To generate the scaffolding for an `article` module based on the `Article` model class, type the following:

```
> symfony propel-generate-crud myapp article Article
```

Symfony reads the definition of the `Article` class in the `schema.yml` and creates a set of templates and actions based on it, in the `myapp/modules/article/` directory.

The generated module contains three views. The `list` view, which is the default view, displays the rows of the `blog_article` table when browsing to `http://localhost/myapp_dev.php/article` as reproduced in Figure 14-2.

article

Id	Title	Content	Created at
1	Welcome to the symfony weblog!	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.	2006-11-12 20:20:25
2	Life is beautiful	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?	2006-11-12 20:20:25

create

Figure 14-2. *`list` view of the `article` module*

Clicking an article identifier displays the `show` view. The details of one row appear in a single page, as in Figure 14-3.

Id:	1
Title:	Welcome to the symfony weblog!
Content:	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.
Created at:	2006-11-12 20:20:25

edit list

Figure 14-3. *`show` view of the `article` module*

Editing an article by clicking the edit link, or creating a new article by clicking the create link in the `list` view, displays the `edit` view, reproduced in Figure 14-4.

Using this module, you can create new articles, and modify or delete existing ones. The generated code is a good base for further developments. Listing 14-2 lists the generated actions and templates of the new module.

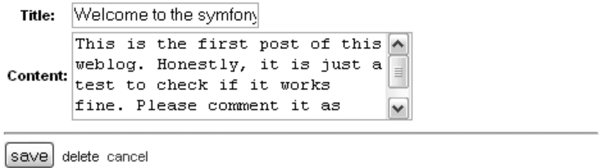


Figure 14-4. *edit view of the article module*

Listing 14-2. *Generated CRUD Elements, in myapp/modules/article/*

```
// In actions/actions.class.php
index           // Forwards to the list action below
list            // Displays the list of all the records of the table
show            // Displays the lists of all columns of a record
edit            // Displays a form to modify the columns of a record
update          // Action called by the edit action form
delete          // Deletes a record
create          // Creates a new record

// In templates/
editSuccess.php  // Record edition form (edit view)
listSuccess.php  // List of all records (list view)
showSuccess.php  // Detail of one record (show view)
```

The logic of these actions and templates is quite simple and explicit, and so rather than reproduce and explain it all, Listing 14-3 gives just a sneak peek on the generated action class.

Listing 14-3. *Generated Action Class, in myapp/modules/article/actions/actions.class.php*

```
class articleActions extends sfActions
{
  public function executeIndex()
  {
    return $this->forward('article', 'list');
  }

  public function executeList()
  {
    $this->articles = ArticlePeer::doSelect(new Criteria());
  }

  public function executeShow()
  {
    $this->article = ArticlePeer::retrieveByPk($this->getRequestParameter('id'));
    $this->forward404Unless($this->article);
  }
  ...
```

Modify the generated code to fit your requirements, repeat the CRUD generation for all the tables that you want to interact with, and you have a basic working application. Generating a scaffolding really bootstraps development; let symfony do the dirty job for you and focus on the interface and specifics.

Initiating a Scaffolding

Initiating a scaffolding is mostly useful when you need to check that you can access the data in the database. It is fast to build and also fast to delete once you're sure that everything works fine.

To initiate a Propel scaffolding that will create an `article` module to deal with the records of the `Article` model class name, type the following:

```
> symfony propel-init-crud myapp article Article
```

You can then access the `list` view using the default action:

```
http://localhost/myapp_dev.php/article
```

The resulting pages are exactly the same as for a generated scaffolding. You can use them as a simple web interface to the database.

If you check the newly created `actions.class.php` in the `article` module, you will see that it is empty: Everything is inherited from an auto-generated class. The same goes for the templates: There is no template file in the `templates/` directory. The code behind the initiated actions and templates is the same as for a generated scaffolding, but lies only in the application cache (`myproject/cache/myapp/prod/module/autoArticle/`).

During application development, developers initiate scaffoldings to interact with the data, regardless of interface. The code is not meant to be customized; an initiated scaffolding can be seen as a simple alternative to PHPmyadmin to manage data.

Administration

Symfony can generate more advanced modules, still based on model class definitions from the `schema.yml` file, for the back-end of your applications. You can create an entire site administration with only generated administration modules. The examples of this section will describe administration modules added to a `backend` application. If your project doesn't have a `backend` application, create its skeleton now by calling the `init-app` task:

```
> symfony init-app backend
```

Administration modules interpret the model by way of a special configuration file called `generator.yml`, which can be altered to extend all the generated components and the module look and feel. Such modules benefit from the usual module mechanisms described in previous chapters (layout, validation, routing, custom configuration, autoloading, and so on). You can also override the generated action or templates, in order to integrate your own features into the generated administration, but `generator.yml` should take care of the most common requirements and restrict the use of PHP code only to the very specific.

Initiating an Administration Module

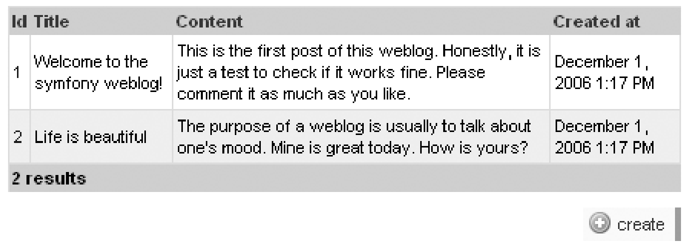
With symfony, you build an administration on a per-module basis. A module is generated based on a Propel object using the `propel-init-admin` task, which uses syntax similar to that used to initiate a scaffolding:

```
> symfony propel-init-admin backend article Article
```

This call is enough to create an `article` module in the `backend` application based on the `Article` class definition, and is accessible by the following:

```
http://localhost/backend.php/article
```

The look and feel of a generated module, illustrated in Figures 14-5 and 14-6, is sophisticated enough to make it usable out of the box for a commercial application.



article list

Id	Title	Content	Created at
1	Welcome to the symfony weblog!	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.	December 1, 2006 1:17 PM
2	Life is beautiful	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?	December 1, 2006 1:17 PM
2 results			

create

Figure 14-5. *`list` view of the `article` module in the `backend` application*

Figure 14-6. *`edit` view of the `article` module in the `backend` application*

The difference between the interface of the scaffolding and the one of the administration may not look significant now, but the configurability of the administration will allow you to enhance the basic layout with many additional features without a line of PHP.

■ **Note** Administration modules can only be initiated (not generated).

A Look at the Generated Code

The code of the `Article` administration module, in the `apps/backend/modules/article/` directory, is empty because it is only initiated. The best way to review the generated code of this module is to interact with it using the browser, and then check the contents of the `cache/` folder. Listing 14-4 lists the generated actions and the templates found in the cache.

Listing 14-4. *Generated Administration Elements, in* `cache/backend/ENV/modules/article/`

```
// In actions/actions.class.php
create           // Forwards to edit
delete           // Deletes a record
edit             // Displays a form to modify the fields of a record
                 // And handles the form submission
index            // Forwards to list
list             // Displays the list of all the records of the table
save             // Forwards to edit

// In templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
editSuccess.php
listSuccess.php
```

This shows that a generated administration module is composed mainly of two views, `edit` and `list`. If you have a look at the code, you will find it to be very modular, readable, and extensible.

Introducing the generator.yml Configuration File

The main difference between scaffoldings and administrations (apart from the fact that administration-generated modules don't have a `show` action) is that an administration relies on parameters found in the `generator.yml` YAML configuration file. To see the default configuration of a newly created administration module, open the `generator.yml` file, located in the `backend/modules/article/config/generator.yml` directory and reproduced in Listing 14-5.

Listing 14-5. *Default Generator Configuration, in `backend/modules/article/config/generator.yml`*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default
```

This configuration is enough to generate the basic administration. Any customization is added under the `param` key, after the `theme` line (which means that all lines added at the bottom of the `generator.yml` file must at least start with four blank spaces to be properly indented). Listing 14-6 shows a typical customized `generator.yml`.

Listing 14-6. *Typical Complete Generator Configuration*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default

    fields:
      author_id:      { name: Article author }

    list:
      title:          List of all articles
      display:        [title, author_id, category_id]
      fields:
        published_on: { params: date_format='dd/MM/yy' }
      layout:         stacked
      params:         |
        %%is_published%%<strong>%%=title%%</strong><br /><em>by %%author%%
        in %%category%% (%%published_on%%)</em><p>%%content_summary%%</p>
      filters:        [title, category_id, author_id, is_published]
      max_per_page:   2
```

```
    edit:
      title:          Editing article "%%title%%"
      display:
        "Post":       [title, category_id, content]
        "Workflow":   [author_id, is_published, created_on]
      fields:
        category_id:  { params: disabled=true }
        is_published: { type: plain}
        created_on:   { type: plain, params: date_format='dd/MM/yy' }
        author_id:    { params: size=5 include_custom=>> Choose an author << }
        published_on: { credentials: [[admin, superdamin]] }
        content:      { params: rich=true tinymce_options=height:150 }
```

The following sections explain in detail all the parameters that can be used in this configuration file.

Generator Configuration

The generator configuration file is very powerful, allowing you to alter the generated administration in many ways. But such capabilities come with a price: The overall syntax description is long to read and learn, making this chapter one of the longest in this book. The symfony website proposes an additional resource that will help you learn this syntax: the administration generator cheat sheet, reproduced in Figure 14-7. Download it from `http://www.symfony-project.com/uploads/assets/sfAdminGeneratorRefCard.pdf`, and keep it close to you when you read the following examples of this chapter.

The examples of this section will tweak the `article` administration module, as well as the `comment` administration module, based on the `Comment` class definition. Create the latter with the `propel-init-admin` task:

```
> symfony propel-init-admin backend comment Comment
```

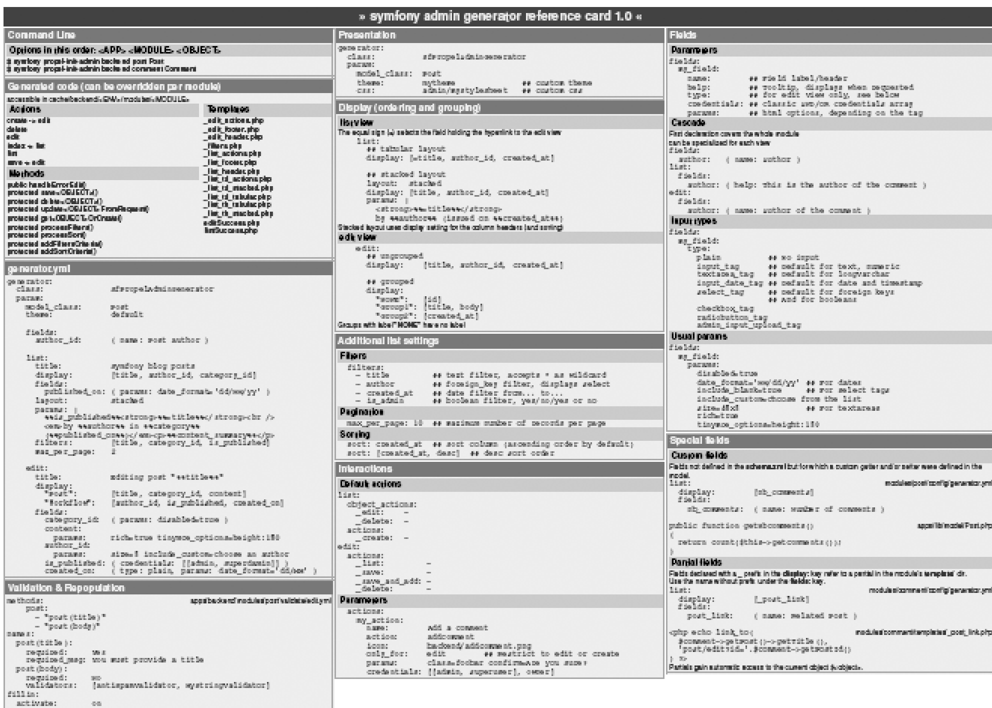


Figure 14-7. *The administration generator cheat sheet*

Fields

By default, the columns of the `list` view and the fields of the `edit` view are the columns defined in `schema.yml`. With `generator.yml`, you can choose which fields are displayed, which ones are hidden, and add fields of your own—even if they don't have a direct correspondence in the object model.

Field Settings

The administration generator creates a `field` for each column in the `schema.yml` file. Under the `fields` key, you can modify the way each field is displayed, formatted, etc. For instance, the field settings shown in Listing 14-7 define a custom label class and input type for the `title` field, and a label and a tooltip for the `content` field. The following sections will describe in detail how each parameter works.

Listing 14-7. *Setting a Custom Label for a Column*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default
```

```
    fields:
      title:          { name: Article Title, type: textarea_tag, params: class=foo }
      content:        { name: Body, help: Fill in the article body }
```

In addition to this default definition for all the views, you can override the field settings for a given view (`list` and `edit`), as demonstrated in Listing 14-8.

Listing 14-8. *Overriding Global Settings View per View*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default

    fields:
      title:          { name: Article Title }
      content:        { name: Body }

    list:
      fields:
        title:        { name: Title }

    edit:
      fields:
        content:      { name: Body of the article }
```

This is a general principle: Any settings that are set for the whole module under the `fields` key can be overridden by view-specific (`list` and `edit`) areas that follow.

Adding Fields to the Display

The fields that you define in the `fields` section can be displayed, hidden, ordered, and grouped in various ways for each view. The `display` key is used for that purpose. For instance, to arrange the fields of the `comment` module, use the code of Listing 14-9.

Listing 14-9. *Choosing the Fields to Display, in `modules/comment/config/generator.yml`*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Comment
    theme:            default

    fields:
      article_id:     { name: Article }
      created_at:     { name: Published on }
      content:        { name: Body }

    list:
      display:        [id, article_id, content]

    edit:
      display:
        NONE:         [article_id]
        Editable:     [author, content, created_at]
```

The `list` will then display three columns, as in Figure 14-8, and the `edit` form will display four fields, assembled in two groups, as in Figure 14-9.

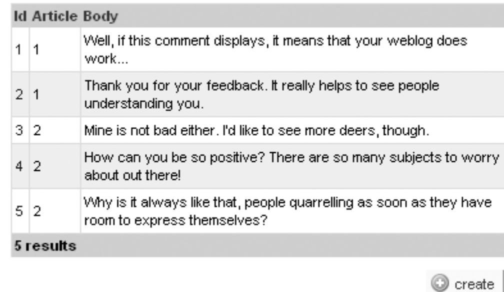


Figure 14-8. *Custom column setting in the `list` view of the `comment` module*

Figure 14-9. *Grouping fields in the `edit` view of the `comment` module*

So you can use the `display` setting in two ways:

- To select the columns to display and the order in which they appear, put the fields in a simple array—as in the previous `list` view.
- To group fields, use an associative array with the group name as a key, or `NONE` for a group with no name. The value is still an array of ordered column names.

■ **Tip** By default, the primary key columns never appear in either view.

Custom Fields

As a matter of fact, the fields configured in `generator.yml` don't even need to correspond to actual columns defined in the schema. If the related class offers a custom getter, it can be used as a field for the `list` view; if there is a getter and/or a setter, it can also be used in the `edit` view. For instance, you can extend the `Article` model with a `getNbComments()` method similar to the one in Listing 14-10.

Listing 14-10. *Adding a Custom Getter in the Model, in `lib/model/Article.class.php`*

```
public function getNbComments()
{
  return $this->countComments();
}
```

Then `nb_comments` is available as a field in the generated module (notice that the getter uses a camelCase version of the field name), as in Listing 14-11.

Listing 14-11. *Custom Getters Provide Additional Columns for Administration Modules, in backend/modules/article/config/generator.yml*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default

    list:
      display:        [id, title, nb_comments, created_at]
```

The resulting `list` view of the `article` module is shown in Figure 14-10.

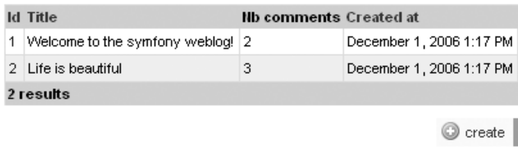


Figure 14-10. *Custom field in the `list` view of the `article` module*

Custom fields can even return HTML code to display more than raw data. For instance, you can extend the `Comment` class with a `getArticleLink()` method as in Listing 14-12.

Listing 14-12. *Adding a Custom Getter Returning HTML, in lib/model/Comment.class.php*

```
public function getArticleLink()
{
  return link_to($this->getArticle()->getTitle(),
                 'article/edit?id='.$this->getArticleId());
}
```

You can use this new getter as a custom field in the `comment/list` view with the same syntax as in Listing 14-11. See the example in Listing 14-13, and the result in Figure 14-11, where the HTML code output by the getter (a hyperlink to the article) appears in the second column instead of the article primary key.

Listing 14-13. *Custom Getters Returning HTML Can Also Be Used As Additional Columns, in modules/comment/config/generator.yml*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Comment
    theme:            default
```

```
list:
  display:        [id, article_link, content]
```

comment list

Id	Article link	Body
1	Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
2	Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
3	Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
4	Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
5	Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results		

create

Figure 14-11. *Custom field in the* `list` *view of the* `comment` *module*

Partial Fields

The code located in the model must be independent from the presentation. The example of the `getArticleLink()` method earlier doesn't respect this principle of layer separation, because some view code appears in the model layer. To achieve the same goal in a correct way, you'd better put the code that outputs HTML for a custom field in a *partial*. Fortunately, the administration generator allows it if you declare a field name prefixed by an underscore. In that case, the `generator.yml` file of Listing 14-13 is to be modified as in Listing 14-14.

Listing 14-14. *Partials Can Be Used As Additional Columns—Use the* `_` *Prefix*

```
list:
  display:        [id, _article_link, created_at]
```

For this to work, an `_article_link.php` partial must be created in the `modules/comment/templates/` directory, as in Listing 14-15.

Listing 14-15. *Example Partial for the* `list` *View, in* `modules/comment/templates/_article_link.php`

```
<?php echo link_to($comment->getArticle()->getTitle(),
                   'article/edit?id='.$comment->getArticleId()) ?>
```

Notice that the partial template of a partial field has access to the current object through a variable named by the class (`$comment` in this example). For instance, for a module built for a class called `UserGroup`, the partial will have access to the current object through the `$user_group` variable.

The result is the same as in Figure 14-11, except that the layer separation is respected. If you get used to respecting the layer separation, you will end up with more maintainable applications.

If you need to customize the parameters of a partial field, do the same as for a normal field, under the `field` key. Just don't include the leading underscore (`_`) in the key—see an example in Listing 14-16.

Listing 14-16. *Partial Field Properties Can Be Customized Under the `fields` Key*

```
fields:
  article_link:   { name: Article }
```

If your partial becomes crowded with logic, you'll probably want to replace it with a component. Change the _ prefix to ~ and you can define a *component field*, as you can see in Listing 14-17.

Listing 14-17. *Components Can Be Used As Additional Columns—Use the ~ Prefix*

```
...
list:
  display:        [id, ~article_link, created_at]
```

In the generated template, this will result by a call to the `articleLink` component of the current module.

■**Note** Custom and partial fields can be used in the `list` view, the `edit` view, and for filters. If you use the same partial for several views, the context (`'list'`, `'edit'`, or `'filter'`) is stored in the `$type` variable.

View Customization

To change the `edit` and `list` views' appearance, you could be tempted to alter the templates. But because they are automatically generated, doing so isn't a very good idea. Instead, you should use the `generator.yml` configuration file, because it can do almost everything that you need without sacrificing modularity.

Changing the View Title

In addition to a custom set of fields, the `list` and `edit` pages can have a custom page title. For instance, if you want to customize the title of the `article` views, do as in Listing 14-18. The resulting `edit` view is illustrated in Figure 14-12.

Listing 14-18. *Setting a Custom Title for Each View, in `backend/modules/article/config/generator.yml`*

```
list:
  title:          List of Articles
  ...
```

```
edit:
  title:          Body of article %%title%%
  display:        [content]
```

Body of article Welcome to the symfony weblog!

Content: This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.

list | save | save and add

delete

Figure 14-12. *Custom title in the* `edit` *view of the* `article` *module*

As the default titles use the class name, they are often good enough—provided that your model uses explicit class names.

■ **Tip** In the string values of `generator.yml`, the value of a field can be accessed via the name of the field surrounded by %%.

Adding Tooltips

In the `list` and `edit` views, you can add tooltips to help describe the fields that are displayed. For instance, to add a tooltip to the `article_id` field of the `edit` view of the `comment` module, add a `help` property in the `fields` definition as in Listing 14-19. The result is shown in Figure 14-13.

Listing 14-19. *Setting a Tooltip in the* `edit` *View, in* `modules/comment/config/generator.yml`

```
edit:
  fields:
    ...
    article_id:   { help: The current comment relates to this article }
```

edit comment

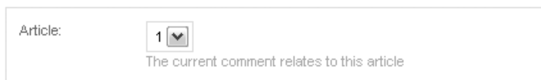


Figure 14-13. *Tooltip in the* `edit` *view of the* `comment` *module*

In the `list` view, tooltips are displayed in the column header; in the `edit` view, they appear under the input.

Modifying the Date Format

Dates can be displayed using a custom format as soon as you use the `date_format` param, as demonstrated in Listing 14-20.

Listing 14-20. *Formatting a Date in the `list` View*

```
list:
  fields:
    created_at:         { name: Published, params: date_format='dd/MM' }
```

It takes the same format parameter as the `format_date()` helper described in the previous chapter.

> **ADMINISTRATION TEMPLATES ARE I18N READY**
>
> All of the text found in the generated templates is automatically internationalized (i.e., enclosed in a call to the `__()` helper). This means that you can easily translate a generated administration by adding the translations of the phrases in an XLIFF file, in your `apps/myapp/i18n/` directory, as explained in the previous chapter.

List View–Specific Customization

The `list` view can display the details of a record in a tabular way, or with all the details stacked in one line. It also contains filters, pagination, and sorting features. These features can be altered by configuration, as described in the next sections.

Changing the Layout

By default, the hyperlink between the `list` view and the `edit` view is borne by the primary key column. If you refer back to Figure 14-11, you will see that the `id` column in the comment list not only shows the primary key of each comment, but also provides a hyperlink allowing users to access the `edit` view.

If you prefer the hyperlink to the detail of the record to appear on another column, prefix the column name by an equal sign (`=`) in the `display` key. Listing 14-21 shows how to remove the `id` from the displayed fields of the comment `list` and to put the hyperlink on the `content` field instead. Check Figure 14-14 for a screenshot.

Listing 14-21. *Moving the Hyperlink for the `edit` View in the `list` View, in `modules/comment/config/generator.yml`*

```
list:
  display:    [article_link, =content]
```

comment list

Article	Body
Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results	

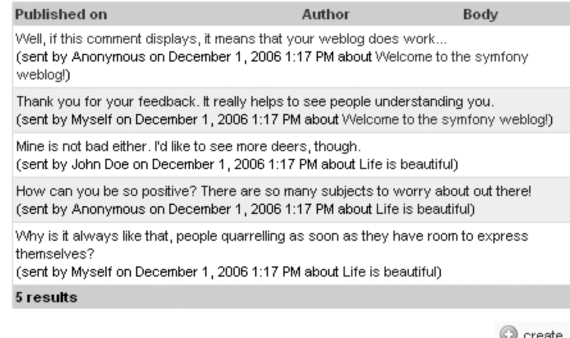
create

Figure 14-14. *Moving the link to the* `edit` *view on another column, in the* `list` *view of the* `comment` *module*

By default, the `list` view uses the `tabular` layout, where the fields appear as columns, as shown previously. But you can also use the `stacked` layout and concatenate the fields into a single string that expands on the full length of the table. If you choose the `stacked` layout, you must set in the `params` key the pattern defining the value of each line of the list. For instance, Listing 14-22 defines a `stacked` layout for the `list` view of the `comment` module. The result appears in Figure 14-15.

Listing 14-22. *Using a* `stacked` *Layout in the* `list` *View, in* `modules/comment/config/generator.yml`

```
list:
  layout:  stacked
  params:  |
    %%=content%% <br />
    (sent by %%author%% on %%created_at%% about %%article_link%%)
  display:  [created_at, author, content]
```



comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
5 results		

create

Figure 14-15. *Stacked layout in the* `list` *view of the* `comment` *module*

Notice that a `tabular` layout expects an array of fields under the `display` key, but a `stacked` layout uses the `params` key for the HTML code generated for each record. However, the `display` array is still used in a `stacked` layout to determine which column headers are available for the interactive sorting.

Filtering the Results

In a `list` view, you can add a set of filter interactions. With these filters, users can both display fewer results and get to the ones they want faster. Configure the filters under the `filters` key, with an array of field names. For instance, add a filter on the `article_id`, `author`, and `created_at` fields to the comment `list` view, as in Listing 14-23, to display a filter box similar to the one in Figure 14-16. You will need to add a `__toString()` method to the `Article` class (returning, for instance, the article `title`) for this to work.

Listing 14-23. *Setting the Filters in the `list` View, in `modules/comment/config/generator.yml`*

```
list:
  filters: [article_id, author, created_at]
  layout:  stacked
  params:  |
    %%=content%% <br />
    (sent by %%author%% on %%created_at%% about %%article_link%%)
  display:  [created_at, author, content]
```

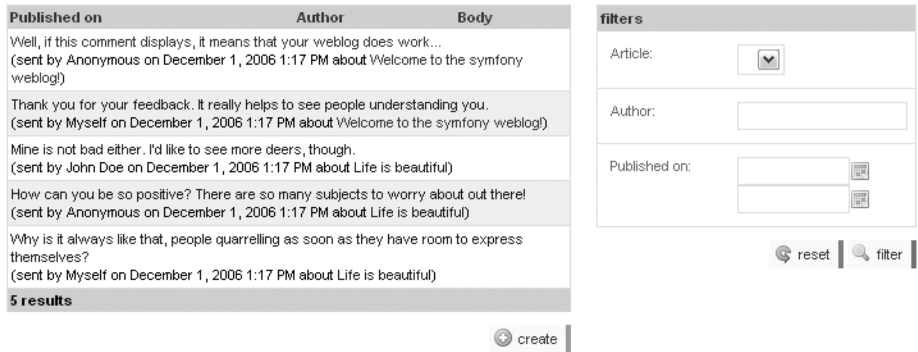


Figure 14-16. *Filters in the `list` view of the `comment` module*

The filters displayed by symfony depend on the column type:

- For text columns (like the `author` field in the `comment` module), the filter is a text input allowing text-based search with wildcards (*).
- For foreign keys (like the `article_id` field in the `comment` module), the filter is a drop-down list of the records of the related table. As for the regular `object_select_tag()`, the options of the drop-down list are the ones returned by the `__toString()` method of the related class.

- For date columns (like the `created_at` field in the `comment` module), the filter is a pair of rich date tags (text fields filled by calendar widgets), allowing the selection of a time interval.
- For Boolean columns, the filter is a drop-down list having `true`, `false`, and `true or false` options—the last value reinitializes the filter.

Just like you use partial fields in lists, you can also use *partial filters* to create a filter that symfony doesn't handle on its own. For instance, imagine a `state` field that may contain only two values (`open` and `closed`), but for some reason you store those values directly in the field instead of using a table relation. A simple filter on this field (of type `string`) would be a text-based search, but what you want is probably a drop-down list of values. That's easy to achieve with a partial filter. See Listing 14-24 for an example implementation.

Listing 14-24. *Using a Partial Filter*

```
// Define the partial, in templates/_state.php
<?php echo select_tag('filters[state]', options_for_select(array(
  '' => '',
  'open' => 'open',
  'closed' => 'closed',
), isset($filters['state']) ? $filters['state'] : '')) ?>

// Add the partial filter in the filter list, in config/generator.yml
    list:
      filters:        [date, _state]
```

Notice that the partial has access to a `$filters` variable, which is useful to get the current value of the filter.

There is one last option that can be very useful for looking for empty values. Imagine that you want to filter the list of comments to display only the ones that have no author. The problem is that if you leave the `author` filter empty, it will be ignored. The solution is to set the `filter_is_empty` field setting to `true`, as in Listing 14-25, and the filter will display an additional check box, which will allow you to look for empty values, as illustrated in Figure 14-17.

Listing 14-25. *Adding Filtering of Empty Values on the `author` Field in the `list` View*

```
    list:
      fields:
        author:   { filter_is_empty: true }
      filters:    [article_id, author, created_at]
```

Figure 14-17. *Allowing the filtering of empty `author` values*

Sorting the List

In a `list` view, the table headers are hyperlinks that can be used to reorder the list, as shown in Figure 14-18. These headers are displayed both in the `tabular` and `stacked` layouts. Clicking these links reloads the page with a `sort` parameter that rearranges the list order accordingly.

List of Articles

Id	Title	Nb comments	Created at
1	Welcome to the symfony weblog!	2	December 1, 2006 1:17 PM
2	Life is beautiful	3	December 1, 2006 1:17 PM
2 results			

create

Figure 14-18. *Table headers of the `list` view are sort controls.*

You can reuse the syntax to point to a list directly sorted according to a column:

```
<?php echo link_to('Comment list by date', 'comment/list?sort=created_at ➥
&type=desc' ) ?>
```

You can also define a default `sort` order for the `list` view directly in the `generator.yml` file. The syntax follows the example given in Listing 14-26.

Listing 14-26. *Setting a Default Sort Field in the `list` View*

```
list:
  sort:   created_at
  # Alternative syntax, to specify a sort order
  sort:   [created_at, desc]
```

■Note Only the fields that correspond to an actual column are transformed into sort controls—not the custom or partial fields.

Customizing the Pagination

The generated administration effectively deals with even large tables, because the `list` view uses pagination by default. When the actual number of rows in a table exceeds the number of maximum rows per page, pagination controls appear at the bottom of the list. For instance, Figure 14-19 shows the list of comments with six test comments in the table but a limit of five comments displayed per page. Pagination ensures a good performance, because only the displayed rows are effectively retrieved from the database, and a good usability, because even tables with millions of rows can be managed by an administration module.

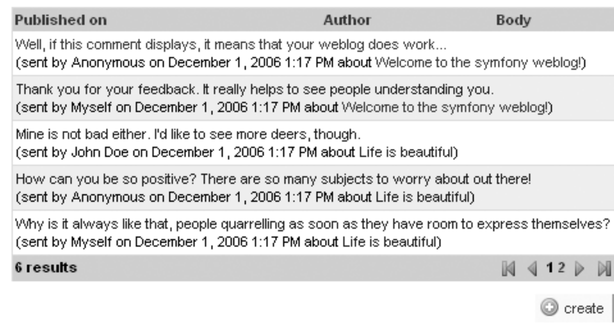


Figure 14-19. *Pagination controls appear on long lists.*

You can customize the number of records to be displayed in each page with the `max_per_page` parameter:

```
list:
  max_per_page:   5
```

Using a Join to Speed Up Page Delivery

By default, the administration generator uses a simple `doSelect()` to retrieve a list of records. But, if you use related objects in the list, the number of database queries required to display the list may rapidly increase. For instance, if you want to display the name of the article in a list of comments, an additional query is required for each post in the list to retrieve the related `Article` object. So you may want to force the pager to use a `doSelectJoinXXX()` method to optimize the number of queries. This can be specified with the `peer_method` parameter.

```
list:
  peer_method:   doSelectJoinArticle
```

Chapter 18 explains the concept of Join more extensively.

Edit View–Specific Customization

In an `edit` view, the user can modify the value of each column for a given record. Symfony determines the type of input to display according to the data type of the column. It then generates

an `object_*_tag` helper, and passes that helper the object and the property to edit. For instance, if the article `edit` view configuration stipulates that the user can edit the `title` field:

```
edit:
  display: [title, ...]
```

then the `edit` page will display a regular text input tag to edit the `title` because this column is defined as a `varchar` type in the schema.

```
<?php echo object_input_tag($article, 'getTitle') ?>
```

Changing the Input Type

The default type-to-field conversion rules are as follows:

- A column defined as `integer`, `float`, `char`, `varchar(size)` appears in the `edit` view as an `object_input_tag()`.
- A column defined as `longvarchar` appears as an `object_textarea_tag()`.
- A foreign key column appears as an `object_select_tag()`.
- A column defined as `boolean` appears as an `object_checkbox_tag()`.
- A column defined as a `timestamp` or `date` appears as an `object_input_date_tag()`.

You may want to override these rules to specify a custom input type for a given field. To that extent, set the `type` parameter in the `fields` definition to a specific form helper name. As for the options of the generated `object_*_tag()`, you can change them with the `params` parameter. See an example in Listing 14-27.

Listing 14-27. *Setting a Custom Input Type and Params for the `edit` View*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Comment
    theme:            default

    edit:
      fields:
                      ## Drop the input, just display plain text
        id:           { type: plain }
                      ## The input is not editable
        author:       { params: disabled=true }
                      ## The input is a textarea (object_textarea_tag)
        content:      { type: textarea_tag,
                        params: rich=true css=user.css tinymce_options=width:330 }
                      ## The input is a select (object_select_tag)
        article_id:   { params: include_custom=Choose an article }
         ...
```

The params parameters are passed as options to the generated object_*_tag(). For instance, the params definition for the preceding article_id will produce in the template the following:

```
<?php echo object_select_tag($comment, 'getArticleId', 'related_class=Article',
           'include_custom=Choose an article') ?>
```

This means that all the options usually available in the form helpers can be customized in an edit view.

Handling Partial Fields

Partial fields can be used in edit views just like in list views. The difference is that you have to handle by hand, in the action, the update of the column according to the value of the request parameter sent by the partial field. Symfony knows how to handle the normal fields (corresponding to actual columns), but can't guess how to handle the inputs you may include in partial fields.

For instance, imagine an administration module for a User class where the available fields are id, nickname, and password. The site administrator must be able to change the password of a user upon request, but the edit view must not display the value of the password field for security reasons. Instead, the form should display an empty password input that the site administrator can fill to change the value. The generator settings for such an edit view are then similar to Listing 14-28.

Listing 14-28. *Including a Partial Field in the edit View*

```
    edit:
      display:         [id, nickname, _newpassword]
      fields:
        newpassword:   { name: Password, help: Enter a password to change it,
                         leave the field blank to keep the current one }
```

The templates/_newpassword.php partial contains something like this:

```
<?php echo input_password_tag('newpassword', '') ?>
```

Notice that this partial uses a simple form helper, not an object form helper, since it is not desirable to retrieve the password value from the current User object to populate the form input—which could disclose the user password.

Now, in order to use the value from this control to update the object in the action, you need to extend the updateUserFromRequest() method in the action. To do that, create a method with the same name in the action class file with the custom behavior for the input of the partial field, as in Listing 14-29.

Listing 14-29. *Handling a Partial Field in the Action, in modules/user/actions/actions.class.php*

```
class userActions extends sfActions
{
  protected function updateUserFromRequest()
  {
    // Handle the input of the partial field
```

```
    $password = $this->getRequestParameter('newpassword');

    if ($password)
    {
      $this->user->setPassword($password);
    }

    // Let symfony handle the other fields
    parent::updateUserFromRequest();
  }
}
```

■ **Note** In the real world, a `user/edit` view usually contains two password fields, the second having to match the first one to avoid typing mistakes. In practice, as you saw in Chapter 10, this is done via a validator. The administration-generated modules benefit from this mechanism just like regular modules.

Dealing with Foreign Keys

If your schema defines table relationships, the generated administration modules take advantage of it and offer even more automated controls, thus greatly simplifying the relationship management.

One-to-Many Relationships

The 1-n table relationships are taken care of by the administration generator. As is depicted by Figure 14-1 earlier, the `blog_comment` table is related to the `blog_article` table through the `article_id` field. If you initiate the module of the `Comment` class with the administration generator, the `comment/edit` action will automatically display the `article_id` as a drop-down list showing the IDs of the available records of the `blog_article` table (check again Figure 14-9 for an illustration).

In addition, if you define a `__toString()` method in the `Article` object, the text of the drop-down options use it instead of the primary keys.

If you need to display the list of comments related to an article in the `article` module (n-1 relationship), you will need to customize the module a little by way of a partial field.

Many-to-Many Relationships

Symfony also takes care of n-n table relationships, but since you can't define them in the schema, you need to add a few parameters to the `generator.yml` file.

The implementation of many-to-many relationships requires an intermediate table. For instance, if there is an n-n relation between a `blog_article` and a `blog_author` table (an article can be written by more than one author and, obviously, an author can write more than one article), your database will always end up with a table called `blog_article_author` or similar, as in Figure 14-20.

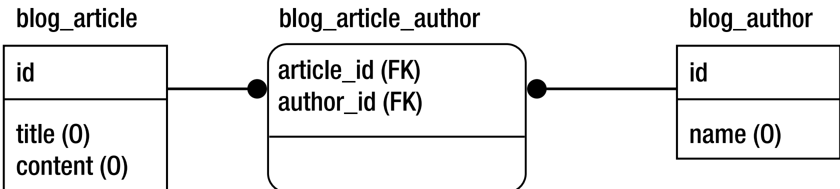


Figure 14-20. *Using a "through class" to implement many-to-many relationships*

The model then has a class called `ArticleAuthor`, and this is the only thing that the administration generator needs—but you have to pass it as a `through_class` parameter of the field.

For instance, in a generated module based on the `Article` class, you can add a field to create new n-n associations with the `Author` class if you write `generator.yml` as in Listing 14-30.

Listing 14-30. *Handling Many-to-Many Relationships with a `through_class` Parameter*

```
edit:
  fields:
    article_author: { type: admin_double_list,
                      params: through_class=ArticleAuthor }
```

Such a field handles links between existing objects, so a regular drop-down list is not enough. You must use a special type of input for that. Symfony offers three widgets to help relate members of two lists (illustrated in Figure 14-21):

- An `admin_double_list` is a set of two expanded select controls, together with buttons to switch elements from the first list (available elements) to the second (selected elements).
- An `admin_select_list` is an expanded select control in which you can select many elements.
- An `admin_check_list` is a list of check box tags.

Figure 14-21. *Available controls for many-to-many relationships*

Adding Interactions

Administration modules allow users to perform the usual CRUD operations, but you can also add your own interactions or restrict the possible interactions for a view. For instance, the interaction definition shown in Listing 14-31 gives access to all the default CRUD actions on the `article` module.

Listing 14-31. *Defining Interactions for Each View, in* `backend/modules/article/config/generator.yml`

```
list:
  title:          List of Articles
  object_actions:
    _edit:         ~
    _delete:       ~
  actions:
    _create:       ~

edit:
  title:          Body of article %%title%%
  actions:
    _list:         ~
    _save:         ~
    _save_and_add: ~
    _delete:       ~
```

In a `list` view, there are two action settings: the list of actions available for every object, and the list of actions available for the whole page. The `list` interactions defined in Listing 14-31 render like in Figure 14-22. Each line shows one button to edit the record and one to delete it. At the bottom of the list, a button allows the creation of a new record.

List of Articles

Id	Title	Nb comments	Created at	Actions
1	Welcome to the symfony weblog!	3	December 1, 2006 1:17 PM	
2	Life is beautiful	3	December 1, 2006 1:17 PM	
2 results				

create

Figure 14-22. *Interactions in the* `list` *view*

In an `edit` view, as there is only one record edited at a time, there is only one set of actions to define. The `edit` interactions defined in Listing 14-31 render like in Figure 14-23. Both the `save` and the `save_and_add` actions save the current edits in the records, the difference being that the `save` action displays the `edit` view on the current record after saving, while the `save_and_add` action displays an empty `edit` view to add another record. The `save_and_add` action is a shortcut that you will find very useful when adding many records in rapid succession. As for the position of the `delete` action, it is separated from the other buttons so that users don't click it by mistake.

The interaction names starting with an underscore (`_`) tell symfony to use the default icon and action corresponding to these interactions. The administration generator understands `_edit`, `_delete`, `_create`, `_list`, `_save`, `_save_and_add`, and `_create`.

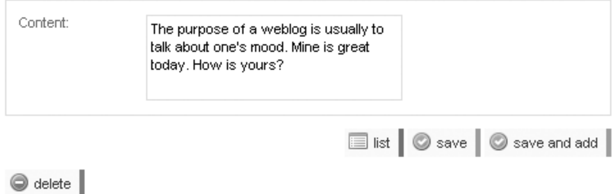


Figure 14-23. *Interactions in the* `edit` *view*

But you can also add a custom interaction, in which case you must specify a name starting with no underscore, as in Listing 14-32.

Listing 14-32. *Defining a Custom Interaction*

```
list:
  title:          List of Articles
  object_actions:
    _edit:        -
    _delete:      -
    addcomment:   { name: Add a comment, action: addComment,
                    icon: backend/addcomment.png }
```

Each article in the list will now show the `addcomment.png` button, as shown in Figure 14-24. Clicking it triggers a call to the `addComment` action in the current module. The primary key of the current object is automatically added to the request parameters.

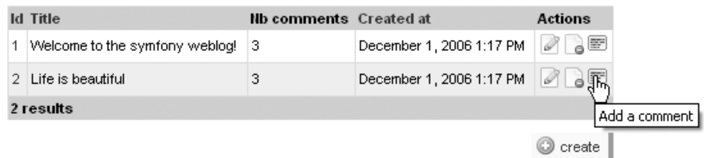


Figure 14-24. *Custom interaction in the* `list` *view*

The `addComment` action can be implemented as in Listing 14-33.

Listing 14-33. *Implementing the Custom Interaction Action, in* `actions/actions.class.php`

```
public function executeAddComment()
{
  $comment = new Comment();
  $comment->setArticleId($this->getRequestParameter('id'));
  $comment->save();
```

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

One last word about actions: If you want to suppress completely the actions for one category, use an empty list, as in Listing 14-34.

Listing 14-34. *Removing All Actions in the* `list` *View*

```
list:
  title:          List of Articles
  actions:        {}
```

Form Validation

If you take a look at the generated `_edit_form.php` template in your project `cache/` directory, you will see that the form fields use a special naming convention. In a generated `edit` view, the input names result from the concatenation of the module name and the field name between angle brackets.

For instance, if the `edit` view for the `article` module has a `title` field, the template will look like Listing 14-35 and the field will be identified as `article[title]`.

Listing 14-35. *Syntax of the Generated Input Names*

```
// generator.yml
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Article
    theme:            default
    edit:
      display: [title]

// Resulting _edit_form.php template
<?php echo object_input_tag($article, 'getTitle', array('control_name' =>
     'article[title]')) ?>

// Resulting HTML
<input type="text" name="article[title]" id="article[title]" value="My Title" />
```

This has plenty of advantages during the internal form-handling process. However, as explained in Chapter 10, it makes the form validation configuration a bit trickier, so you have to change square brackets, [ ], to curly braces, { }, in the `fields` definition. Also, when using a field name as a parameter for a validator, you should use the name as it appears in the generated HTML code (that is, with the square brackets, but between quotes). Refer to Listing 14-36 for a detail of the special validator syntax for generated forms.

Listing 14-36. *Validator File Syntax for Administration-Generated Forms*

```
## Replace square brackets by curly brackets in the fields list
fields:
  article{title}:
    required:
      msg: You must provide a title
    ## For validator parameters, use the original field name between quotes
    sfCompareValidator:
      check:        "user[newpassword]"
      compare_error: The password confirmation does not match the password.
```

Restricting User Actions Using Credentials

For a given administration module, the available fields and interactions can vary according to the credentials of the logged user (refer to Chapter 6 for a description of symfony's security features).

The fields in the generator can take a `credentials` parameter into account so as to appear only to users who have the proper credential. This works for the `list` view and the `edit` view. Additionally, the generator can also hide interactions according to credentials. Listing 14-37 demonstrates these features.

Listing 14-37. *Using Credentials in `generator.yml`*

```
## The id column is displayed only for users with the admin credential
    list:
      title:          List of Articles
      layout:         tabular
      display:        [id, =title, content, nb_comments]
      fields:
        id:           { credentials: [admin] }

## The addcomment interaction is restricted to the users with the admin credential
    list:
      title:          List of Articles
      object_actions:
        _edit:        -
        _delete:      -
        addcomment:   { credentials: [admin], name: Add a comment,
                        action: addComment, icon: backend/addcomment.png }
```

Modifying the Presentation of Generated Modules

You can modify the presentation of the generated modules so that it matches any existing graphical charter, not only by applying your own style sheet, but also by overriding the default templates.

Using a Custom Style Sheet

Since the generated HTML is structured content, you can do pretty much anything you like with the presentation.

You can define an alternative CSS to be used for an administration module instead of a default one by adding a `css` parameter to the generator configuration, as in Listing 14-38.

Listing 14-38. *Using a Custom Style Sheet Instead of the Default One*

```
generator:
  class:              sfPropelAdminGenerator
  param:
    model_class:      Comment
    theme:            default
    css:              mystylesheet
```

Alternatively, you can also use the mechanisms provided by the module `view.yml` to override the styles on a per-view basis.

Creating a Custom Header and Footer

The `list` and `edit` views systematically include a header and footer partial. There is no such partial by default in the `templates/` directory of an administration module, but you just need to add one with one of the following names to have it included automatically:

```
_list_header.php
_list_footer.php
_edit_header.php
_edit_footer.php
```

For instance, if you want to add a custom header to the `article/edit` view, create a file called `_edit_header.php` as in Listing 14-39. It will work with no further configuration.

Listing 14-39. *Example `edit` Header Partial, in `modules/articles/template/_edit_header.php`*

```
<?php if ($article->getNbComments() > 0): ?>
  <h2>This article has <?php echo $article->getNbComments() ?> comments.</h2>
<?php endif; ?>
```

Notice that an `edit` partial always has access to the current object through a variable having the same name as the module, and that a `list` partial always has access to the current pager through the `$pager` variable.

CALLING THE ADMINISTRATION ACTIONS WITH CUSTOM PARAMETERS

The administration module actions can receive custom parameters using the `query_string` argument in a `link_to()` helper. For example, to extend the previous `_edit_header` partial with a link to the comments for the article, write this:

```
<?php if ($article->getNbComments() > 0): ?>
  <h2>This article has
  <?php echo link_to($article->getNbComments().' comments', 'comment/list',
        array('query_string' => 'filter=filter&filters%5Barticle_id%5D='.
        $article->getId())) ?></h2>
<?php endif; ?>
```

This query string parameter is an encoded version of the more legible

```
'filter=filter&filters[article_id]='.$article->getId()
```

It filters the comments to display only the ones related to `$article`. Using the `query_string` argument, you can specify a sorting order and/or a filter to display a custom `list` view. This can also be useful for custom interactions.

Customizing the Theme

There are other partials inherited from the framework that can be overridden in the module `templates/` folder to match your custom requirements.

The generator templates are cut into small parts that can be overridden independently, and the actions can also be changed one by one.

However, if you want to override those for several modules in the same way, you should probably create a reusable *theme*. A theme is a complete set of templates and actions that can be used by an administration module if specified in the `theme` value at the beginning of `generator.yml`. With the `default` theme, symfony uses the files defined in `$sf_symfony_data_dir/generator/sfPropelAdmin/default/`.

The theme files must be located in a project tree structure, in a `data/generator/sfPropelAdmin/[theme_name]/template/` directory, and you can bootstrap a new theme by copying the files from the `default` theme (located in `$sf_symfony_data_dir/generator/sfPropelAdmin/default/template/` directory). This way, you are sure that all the files required for a theme will be present in your custom theme:

```
// Partials, in [theme_name]/template/templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
```

```
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php

// Actions, in [theme_name]/template/actions/actions.class.php
processFilters()     // Process the request filters
addFiltersCriteria() // Adds a filter to the Criteria object
processSort()
addSortCriteria()
```

Be aware that the template files are actually *templates of templates*, that is, PHP files that will be parsed by a special utility to generate templates based on generator settings (this is called the *compilation* phase). The generated templates must still contain PHP code to be executed during actual browsing, so the templates of templates use an alternative syntax to keep PHP code unexecuted for the first pass. Listing 14-40 shows an extract of a `default` template of template.

Listing 14-40. *Syntax of Templates of Templates*

```
<?php foreach ($this->getPrimaryKey() as $pk): ?>
[?php echo object_input_hidden_tag($<?php echo $this->getSingularName() ?>, ➥
 'get<?php echo $pk->getPhpName() ?>') ?]
<?php endforeach; ?>
```

In this listing, the PHP code introduced by `<?` is executed immediately (at compilation), the one introduced by `[?` is only executed at execution, but the templating engine finally transforms the `[?` tags into `<?` tags so that the resulting template looks like this:

```
<?php echo object_input_hidden_tag($article, 'getId') ?>
```

Dealing with templates of templates is tricky, so the best advice if you want to create your own theme is to start from the `default` theme, modify it step by step, and test it extensively.

■ **Tip** You can also package a generator theme in a plug-in, which makes it even more reusable and easy to deploy across multiple applications. Refer to Chapter 17 for more information.

BUILDING YOUR OWN GENERATOR

The scaffolding and administration generators both use a set of symfony internal components that automate the creation of generated actions and templates in the cache, the use of themes, and the parsing of templates of templates.

This means that symfony provides all the tools to build your own generator, which can look like the existing ones or be completely different. The generation of a module is managed by the `generate()` method of the `sfGeneratorManager` class. For instance, to generate an administration, symfony calls the following internally:

```
$generator_manager = new sfGeneratorManager();
$data = $generator_manager->generate('sfPropelAdminGenerator', $parameters);
```

If you want to build your own generator, you should look at the API documentation of the `sfGeneratorManager` and the `sfGenerator` classes, and take as examples the `sfAdminGenerator` and `sfCRUDGenerator` classes.

Summary

To bootstrap your modules or automatically generate your back-end applications, the basis is a well-defined *schema and object model*. You can modify the PHP code of *scaffoldings*, but *administration-generated* modules are to be modified mostly through configuration.

The `generator.yml` file is the heart of the programming of generated back-ends. It allows for the complete customization of content, features, and the look and feel of the `list` and `edit` views. You can manage *field labels*, *tooltips*, *filters*, *sort order*, *page size*, *input type*, *foreign relationships*, *custom interactions*, and *credentials* directly in YAML, without a single line of PHP code.

If the administration generator doesn't natively support the feature you need, the *partial fields* and the *ability to override actions* provide complete extensibility. Plus, you can reuse your adaptations to the administration generator mechanisms thanks to the *theme* mechanisms.