



My first symfony project

This PDF is brought to you by
SENSIOLABS 

License: Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License
Version: my-first-project-1.2-en-2009-12-05

Table of Contents

Array

My first symfony project

So, you want to give it a go? Let's build together a fully-functional web app in one hour. You name it. A bookseller application? Okay, another idea. A blog! That's a good one. Let's go.

This tutorial assumes that you are working with Apache installed and running on your local machine. You will also need PHP 5.1.3 or newer.

Install symfony and initialize the project

To go fast, the symfony sandbox will be used. This is an empty symfony project where all the required libraries are already included, and where the basic configuration is already done. The great advantage of the sandbox over other types of installation is that you can start experimenting with symfony immediately.

Get it here: [sf_sandbox_1_2.tgz](#)¹ or here: [sf_sandbox_1_2.zip](#)², and unpack it in your root web directory. On Linux systems, it is recommended to keep the permissions as they are in the tar file (for example by using `-p` with `tar` command). Refer to the included README file for more information. The resulting file structure should look like this:

```
www/  
  sf_sandbox/  
    apps/  
      frontend/  
    cache/  
    config/  
    data/  
    doc/  
    lib/  
    log/  
    plugins/  
    test/  
    web/  
      css/  
      images/  
      js/
```

Listing
1-1

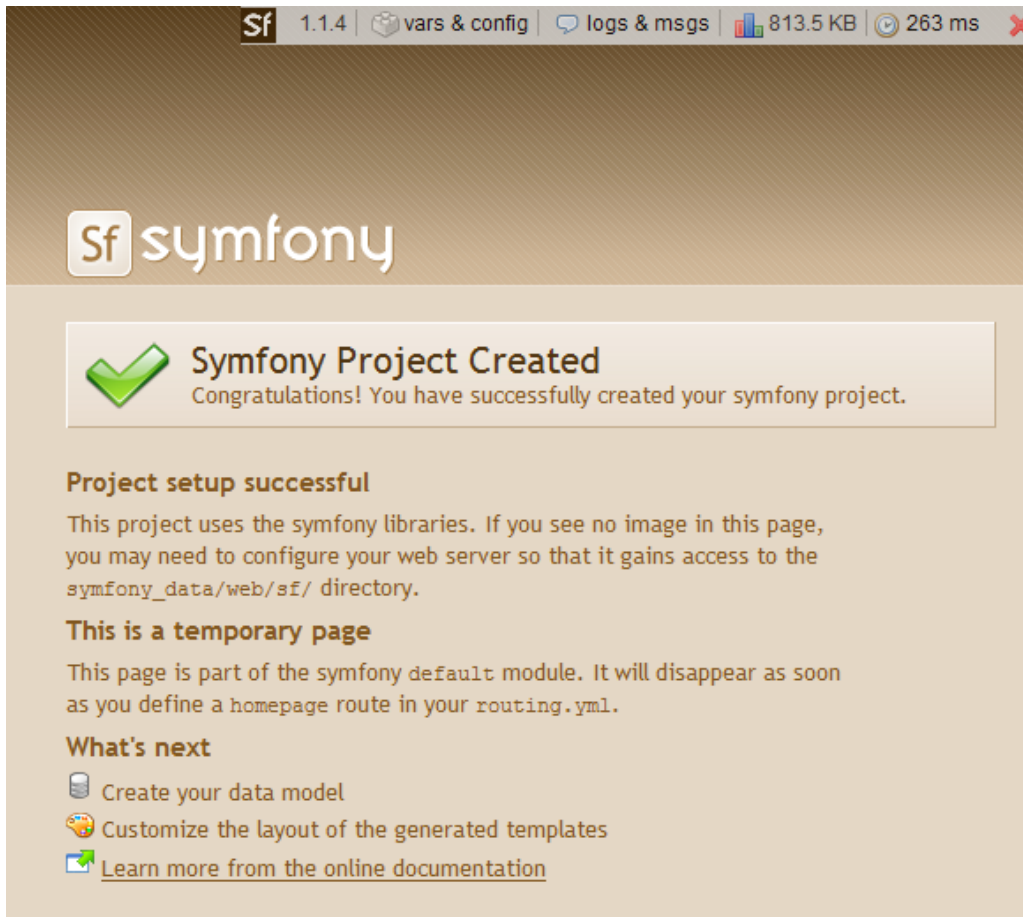
This shows a `sf_sandbox` **project** containing a frontend **application**. Test the sandbox by requesting the following URL:

```
http://localhost/sf_sandbox/web/index.php/
```

Listing
1-2

You should see a congratulations page.

1. http://www.symfony-project.org/get/sf_sandbox_1_2.tgz
2. http://www.symfony-project.org/get/sf_sandbox_1_2.zip



You can also install symfony in a custom folder and setup your web server with a Virtual Host or an Alias. The symfony book contains detailed chapters about symfony installation³ and the symfony directory structure⁴.

Initialize the data model

So, the blog will handle posts, and you will enable comments on them. Create a `schema.yml` file in `sf_sandbox/config/` and paste the following data model:

Listing
1-3

```
propel:
  blog_post:
    id: ~
    title: { type: varchar(255), required: true }
    excerpt: { type: longvarchar }
    body: { type: longvarchar }
    created_at: ~
  blog_comment:
    id: ~
    blog_post_id: ~
    author: { type: varchar(255) }
    email: { type: varchar(255) }
    body: { type: longvarchar }
    created_at: ~
```

3. http://www.symfony-project.org/book/1_2/03-Running-Symfony

4. http://www.symfony-project.org/book/1_2/02-Exploring-Symfony-s-Code

This configuration file uses the YAML syntax. It's a very simple language that allows XML-like tree structures described by indentation. Furthermore, it is faster to read and write than XML. The only thing is, the indentation has a meaning and tabulations are forbidden, so remember to use spaces for indentation. You will find more about YAML and the symfony configuration in the configuration chapter⁵.

This schema describes the structure of two of the tables needed for the blog. `blog_post` and `blog_comment` are the names of the related classes to be generated. Save the file, open a command line, browse to the `sf_sandbox/` directory and type:

```
$ php symfony propel:build-model
```

Listing
1-4



Make sure your command line folder is set to the root of your project (`sf_sandbox/`) when you call the `symfony` command.

If you receive the error 'Could not perform XSLT transformation', check that you have the `php_xsl` extension enabled in your `php.ini` file.

A few classes are created in the `sf_sandbox/lib/model/` directory. These are the classes of the object-relational mapping system, which allows us to have access to a relational database from within an object-oriented code without writing a single SQL query. By default, symfony uses the Propel library for this purpose. These classes are part of the **model** of our application (find more in the model chapter⁶).

Now, we need to convert the schema to SQL statements to initialize the database tables. By default, the symfony sandbox is configured to work out of the box with a simple SQLite file, so no database initialization is required. You still need to check that the SQLite extension is installed and enabled correctly (you can check this in `php.ini` - see how in the PHP documentation⁷).

By default, the `sf_sandbox` project will use a database file called `sandbox.db` located in `sf_sandbox/data/`.

If you want to switch to MySQL for this project, use the `configure:database` task:

```
$ php symfony configure:database
"mysql:dbname=symfony_project;host=localhost" root mypassword
```

Listing
1-5

Just ensure to have a `symfony_project` mysql database available, accessible by the user `root` using the `mypassword` password.

Change the DSN argument to match your settings (username, password, host, and database name) and then create the database with the command line or a web interface (as described in the model chapter⁸). Then, open `sf_sandbox/config/databases.yml` and set 'phptype' to 'mysql', and 'database' to the name of your MySQL database.



If you use SQLite as your database engine, you will have to change some rights on *nix systems:

```
$ chmod 777 data data/sandbox.db
```

Listing
1-6

Now type in the command line:

5. http://www.symfony-project.org/book/1_2/05-Configuring-Symfony
6. http://www.symfony-project.org/book/1_2/08-Inside-the-Model-Layer
7. <http://fr3.php.net/manual/en/ref.sqlite.php>
8. http://www.symfony-project.org/book/1_2/08-Inside-the-Model-Layer

Listing 1-7

```
$ php symfony propel:build-sql
```

A `lib.model.schema.sql` file is created in `sf_sandbox/data/sql/`. The SQL statements found in this file can be used to initialize a database with the same table structure.

To build the table structure based on the the SQL file, type:

Listing 1-8

```
$ php symfony propel:insert-sql
```



Don't worry if there is a warning at that point, it is normal. The `propel:insert-sql` command removes existing tables before adding the ones from your `lib.model.schema.sql`, and there are no tables to remove at the moment.

As you want to be able to create and edit blog posts and comments, you also need to generate some forms based on the model schema:

Listing 1-9

```
$ php symfony propel:build-forms
```

This task generates classes in the `sf_sandbox/lib/form/` directory. These classes are used to manage your model objects as forms.



All the above commands can be done in a single one by using `propel:build-all`.

Create the application

The basic features of a blog are to be able to Create, Retrieve, Update and Delete (CRUD) posts and comments. As you are new to symfony, you will not create symfony code from scratch, but rather let it generate the code that you may use and modify as needed. Symfony can interpret the data model to generate the CRUD interface automatically:

Listing 1-10

```
$ php symfony propel:generate-module --non-verbose-templates --with-show  
frontend post BlogPost  
$ php symfony propel:generate-module --non-verbose-templates frontend  
comment BlogComment  
$ php symfony cache:clear
```



When using the `propel:generate-module` task, you have used the `--non-verbose-templates` option. If you want to learn the meaning of the available arguments and options for a given task, you can use the special `help` task:

Listing 1-11

```
$ php symfony help propel:generate-module
```

You now have two modules (`post` and `comment`) that will let you manipulate objects of the `BlogPost` and `BlogComment` classes. A **module** usually represents a page or a group of pages with a similar purpose. Your new modules are located in the `sf_sandbox/apps/frontend/modules/` directory, and they are accessible by the URLs:

Listing 1-12

```
http://localhost/sf_sandbox/web/frontend_dev.php/post  
http://localhost/sf_sandbox/web/frontend_dev.php/comment
```

If you try to create a comment, you will have an error because symfony doesn't yet know how to convert a post object to a string. Edit the `BlogPost` class (`lib/model/BlogPost.php`) and add the `__toString()` method:

```
class BlogPost extends BaseBlogPost
{
    public function __toString()
    {
        return $this->getTitle();
    }
}
```

Listing
1-13

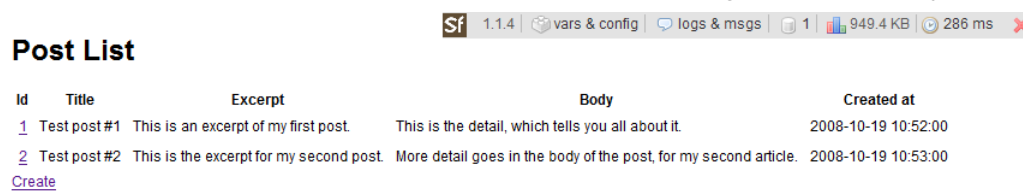
Lastly, add the following CSS to `sf_sandbox/web/css/main.css`:

```
body, td
{
    font-family: Arial, Verdana, sans-serif;
    font-size: 12px;
}

td { margin: 4px; padding: 4px; }
```

Listing
1-14

Now, feel free to create some new posts to make the blog look less empty.



Id	Title	Excerpt	Body	Created at
1	Test post #1	This is an excerpt of my first post.	This is the detail, which tells you all about it.	2008-10-19 10:52:00
2	Test post #2	This is the excerpt for my second post.	More detail goes in the body of the post, for my second article.	2008-10-19 10:53:00

[Create](#)

Find more about generators⁹ and the explanation of symfony projects structure¹⁰ (project, application, module).



In the URLs above, the name of the main script - called the *front controller* in symfony - was changed from `index.php` to `frontend_dev.php`. The two scripts access the same application (frontend), but in different environments. With `frontend_dev.php`, you access the application in the **development environment**, which provides handy development tools like the debug toolbar on the top right of the screen and the live configuration engine. That's why the processing of each page is slower than when using `index.php`, which is the front controller of the **production environment**, optimized for speed. If you want to keep on using the production environment, replace `frontend_dev.php/` by `index.php/` in the following URLs, but don't forget to clear the cache before watching the changes:

```
$ php symfony cache:clear

http://localhost/sf_sandbox/web/index.php/
```

Listing
1-15

Find more about environments¹¹.

9. http://www.symfony-project.org/book/1_2/14-Generators

10. http://www.symfony-project.org/book/1_2/04-The-Basics-of-Page-Creation

11. http://www.symfony-project.org/book/1_2/05-Configuring-Symfony#Environments

Modify the layout

In order to navigate between the two new modules, the blog needs some global navigation.

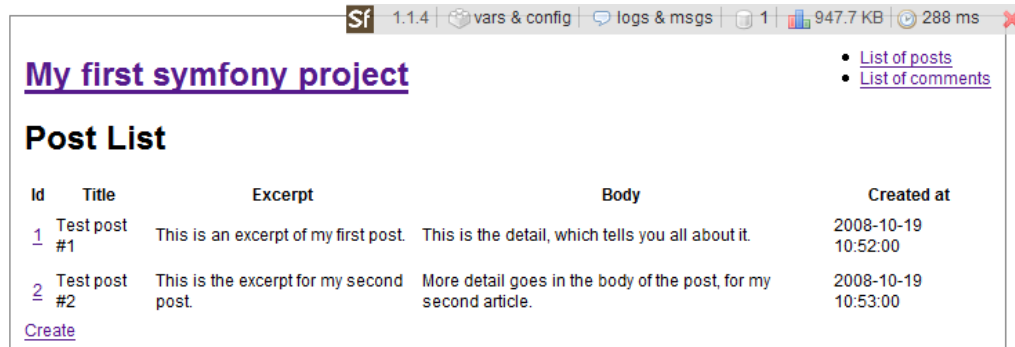
Edit the global template `sf_sandbox/apps/frontend/templates/layout.php` and change the content of the `<body>` tag to:

Listing
1-16

```
<div id="container" style="width:700px;margin:0 auto;border:1px solid
grey;padding:10px">
  <div id="navigation" style="display:inline;float:right">
    <ul>
      <li><?php echo link_to('List of posts', 'post/index') ?></li>
      <li><?php echo link_to('List of comments', 'comment/index') ?></li>
    </ul>
  </div>
  <div id="title">
    <h1><?php echo link_to('My first symfony project', '@homepage') ?></h1>
  </div>

  <div id="content" style="clear:right">
    <?php echo $sf_data->getRaw('sf_content') ?>
  </div>
</div>
```

Please forgive the poor design and the use of inner-tag css, but one hour is rather a short amount of time!



While you are at it, you can change the title of your pages. Edit the view configuration file of the application (`sf_sandbox/apps/frontend/config/view.yml`), locate the line showing the `title` key and change it something appropriate. Note that several lines here are commented out with a hash symbol - you can uncomment these if you wish.

Listing
1-17

```
default:
  http_metas:
    content-type: text/html

  metas:
    title:           The best blog ever
    #description:    symfony project
    #keywords:       symfony, project
    #language:       en
    robots:         index, follow
```

The home page itself needs to be changed. It uses the default template of the default module, which is kept in the framework but not in your application directory. To override it, you can create a custom main module:

```
$ php symfony generate:module frontend main
```

Listing
1-18

By default, the `index` action shows a default congratulations screen. To remove it, edit the `sf_sandbox/apps/frontend/modules/main/actions/actions.class.php` and remove the content of the `executeIndex()` method as follows:

```
/**
 * Executes index action
 *
 * @param sfRequest $request A request object
 */
public function executeIndex(sfWebRequest $request)
{
}
```

Listing
1-19

Edit the `sf_sandbox/apps/frontend/modules/main/templates/indexSuccess.php` file to show a nice welcome message:

```
<h1>Welcome to my new blog</h1>
<p>You are the <?php echo rand(1000,5000) ?>th visitor today.</p>
```

Listing
1-20

Now, you must tell symfony which action to execute when the homepage is requested. To do so, edit the `sf_sandbox/apps/frontend/config/routing.yml` and change the homepage rule as follows:

```
homepage:
  url: /
  param: { module: main, action: index }
```

Listing
1-21

Check the result by requesting the home page again:

```
http://localhost/sf_sandbox/web/frontend_dev.php/
```

Listing
1-22

Go ahead, start using your new web app. Make sure you've created a test post, and also create a test comment against your post.

Find more about views and templates¹².

Pass data from the action to the template

That was fast, wasn't it? Now it is time to mix the comment module into the post one to get comments displayed below posts.

First, you need to make the post comments available for the post display template. In symfony, this kind of logic is kept in **actions**. Edit the actions file `sf_sandbox/apps/frontend/modules/post/actions/actions.class.php` and change the `executeShow()` method by adding the four last lines:

12. http://www.symfony-project.org/book/1_2/07-Inside-the-View-Layer

Listing
1-23

```
public function executeShow(sfWebRequest $request)
{
    $this->blog_post =
BlogPostPeer::retrieveByPk($request->getParameter('id'));
    $this->forward404Unless($this->blog_post);

    $c = new Criteria();
    $c->add(BlogCommentPeer::BLOG_POST_ID, $request->getParameter('id'));
    $c->addAscendingOrderByColumn(BlogCommentPeer::CREATED_AT);
    $this->comments = BlogCommentPeer::doSelect($c);
}
```

The Criteria and -Peer objects are part of Propel's object-relational mapping. Basically, these four lines will handle a SQL query to the blog_comment table to get the comments related to the current blog_post. Now, modify the post display template sf_sandbox/apps/frontend/modules/post/templates/showSuccess.php by adding at the end:

Listing
1-24

```
// ...
<?php use_helper('Text', 'Date') ?>

<hr />
<?php if ($comments) : ?>
    <p><?php echo count($comments) ?> comments to this post.</p>
    <?php foreach ($comments as $comment): ?>
        <p><em>posted by <?php echo $comment->getAuthor() ?> on <?php echo
format_date($comment->getCreatedAt()) ?></em></p>
        <div class="comment" style="margin-bottom:10px;">
            <?php echo simple_format_text($comment->getBody()) ?>
        </div>
    <?php endforeach; ?>
<?php endif; ?>
```

This page uses new PHP functions that are called 'helpers', because they do some tasks for you that would normally require more time and code. Create a new comment for your first post, then check again the first post, either by clicking on its number in the list, or by typing directly:

Listing
1-25

http://localhost/sf_sandbox/web/frontend_dev.php/post/show?id=1

My first symfony project

- List of posts
- List of comments

Id: 2
Title: Test post #2
Excerpt: This is the excerpt for my second post.
Body: More detail goes in the body of the post, for my second article.
Created at: 2008-10-19 10:53:00

[Edit](#) [List](#)

2 comments to this post

posted by Jon on 2008-10-19

I agree. This is an excellent post.

posted by Jon on 2008-10-19

This is a test comment

This is getting good.

Find more about the naming conventions¹³ linking an action to a template.

Add a record relative to another table

Currently you can't add comments to posts directly; if you edit a post, you have to go to the comments editing section, create a new one, then select the post you want to comment on using the drop-down menu. The screen looks like this:

It would be better if there was a link on each post editing page to go straight to the comment editing facility, so let's arrange that first. In the `sf_sandbox/apps/frontend/modules/post/templates/showSuccess.php` template, add this line at the bottom:

```
<?php echo link_to('Add a comment', 'comment/new?blog_post_id='.$blog_post->getId()) ?>
```

Listing
1-26

The `link_to()` helper creates a hyperlink pointing to the edit action of the comment module, so you can add a comment directly from the post details page. At the moment, however, the comments edit page still offers a form element to select which post to relate a comment to. This would be best replaced by a hidden field (containing the post primary key) if the comments edit page URL is called specifying that key.

In symfony, forms are managed by classes. So, let's edit the `BlogCommentForm` class to make our changes. The file is located under the `sf_sandbox/lib/form/` directory:

```
class BlogCommentForm extends BaseBlogCommentForm
{
    /**
     * Configure method, called when the form is instantiated
     */
    public function configure()
    {
        $this->widgetSchema['blog_post_id'] = new sfWidgetFormInputHidden();
    }
}
```

Listing
1-27

13. http://www.symfony-project.org/book/1_2/07-Inside-the-View-Layer



For more details on the form framework, you are advised to read the Forms Book¹⁴.

Then, the post id has to be automatically set from the request parameter passed in the URL. This can be done by replacing the `executeNew()` method of the `comment` module by the following one.

Listing
1-28

```
class commentActions extends sfActions
{
    public function executeNew(sfWebRequest $request)
    {
        $this->form = new BlogCommentForm();
        $this->form->setDefault('blog_post_id',
$request->getParameter('blog_post_id'));
    }

    // ...
}
```

After you have made this change, you will now be able to add a comment directly to a post without having to explicitly specify the post to attach it to:

Next, after adding a comment, we want the user to come back to the post it relates to, rather than remaining on the comment editing page. To accomplish this, we need to edit `processForm` method. Find the following code in `sf_sandbox/apps/frontend/modules/comment/actions/actions.class.php`:

Listing
1-29

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

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

14. http://www.symfony-project.org/book/forms/1_2/en/

And change the redirect line so it reads thus:

```
$this->redirect('post/show?id='.$blog_comment->getBlogPostId());
```

*Listing
1-30*

This will ensure that when a comment is saved, the user is returned to the post that the comment is related to. There are two things here that are worthy of note: firstly, the save is achieved simply by calling the save method on the form object (this is because the form is associated with the Propel model and therefore knows how to serialize the object back to the database). Secondly, we redirect immediately after the save, so that if the page is subsequently refreshed, the user is not asked if they wish to repeat the POST action again.

Okay, so that wraps up this part of the tutorial. You wanted a blog? You have a blog. Incidentally, since we've covered symfony actions a lot here, you may wish to find out more about them from the manual¹⁵.

Form Validation

Visitors can enter comments, but what if they submit the form without any data in it, or data that is obviously wrong? You would end up with a database containing invalid rows. To avoid that, we need to set up some validation rules to specify what data is allowed.

When symfony has created the form classes for us, it has generated the form elements to render on the screen, but it has also added some default validation rules by introspecting the schema. As the `title` is required in the `blog_post` table, you won't be able to submit a form without a title. You also won't be able to submit a post with a title longer than 255 character.

Let's override some of these rules now in the `BlogCommentForm` class. So, open the file, and add in the following PHP code at the end of the `configure()` method:

```
$this->validatorSchema['email'] = new sfValidatorEmail(
    array('required' => false),
    array('invalid' => 'The email address is not valid'));
```

*Listing
1-31*

By redefining the validator for the `email` column, we have overridden the default behavior.

Once this new rule is in place, try saving a comment with a bad email address - you now have a robust form! You will notice a number of things: first of all, where the form contains data, that data will automatically be preserved during the form submission. This saves the user having to type it back in (and normally is something in that the programmer has to arrange manually). Also, errors (in this case) are placed next to the fields that failed their associated validation tests.

Now would be a good time to explain a little about how the form save process works. It uses the following action code, which you edited earlier in `/sf_sandbox/apps/frontend/modules/comment/actions/actions.class.php`:

```
$this->form = new
BlogCommentForm(BlogCommentPeer::retrieveByPk($request->getParameter('id')));

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

*Listing
1-32*

15. http://www.symfony-project.org/book/1_2/06-Inside-the-Controller-Layer

```

    $this->redirect('post/show?id='.$blog_comment->getBlogPostId());
}
}

```

After the form object is instantiated, the following happens:

- The code checks that the HTTP method is a POST
- The parameter array `blog_comment` is retrieved. The `getParameter()` method detects that this name is an array of values in the form, not a single value, and returns them as an associative array (e.g. form element `blog_comment[author]` is returned in an array having a key of `author`)
- This associative array is then fed into the form using a process called **binding**, in which the values are used to fill form elements in the form object. After this, the values are determined to have either passed or failed the validation checks
- Only if the form is valid does the save go ahead, after which the page redirects immediately to the show action.

The screenshot shows a web browser window titled 'My first symfony project'. The page content includes a header with navigation links for 'List of posts' and 'List of comments'. Below the header is a form titled 'New Comment'. The form contains several fields with associated validation messages:

- Blog post id:** A dropdown menu with a message: 'This comment must have an associated post'.
- Author:** A text input field with a message: 'The author field cannot be left blank'.
- Email:** A text input field containing 'bad_email@' with a message: 'The email address is not valid'.
- Body:** A large text area with a message: 'The body field cannot be left blank'.
- Created at:** A date and time selector with dropdown menus.

At the bottom of the form are 'Cancel' and 'Save' buttons.

Find more about form validation¹⁶.

Change the URL format

Did you notice the way the URLs are rendered? You can make them more user and search engine-friendly. Let's use the post title as a URL for posts.

The problem is that post titles can contain special characters like spaces. If you just escape them, the URL will contain some ugly `%20` strings, so the model needs to be extended with a new method in the `BlogPost` object, to get a clean, stripped title. To do that, edit the file `BlogPost.php` located in the `sf_sandbox/lib/model/` directory, and add the following method:

16. http://www.symfony-project.org/book/forms/1_2/en/02-Form-Validation

```

public function getStrippedTitle()
{
    $result = strtolower($this->getTitle());

    // strip all non word chars
    $result = preg_replace('/\W/', ' ', $result);

    // replace all white space sections with a dash
    $result = preg_replace('/\ +/', '-', $result);

    // trim dashes
    $result = preg_replace('/\-$/', '', $result);
    $result = preg_replace('/^\-/', '', $result);

    return $result;
}

```

*Listing
1-33*

Now you can create a permalink action for the post module. Add the following method to the `sf_sandbox/apps/frontend/modules/post/actions/actions.class.php`:

```

public function executePermalink($request)
{
    $posts = BlogPostPeer::doSelect(new Criteria());
    $title = $request->getParameter('title');
    foreach ($posts as $post)
    {
        if ($post->getStrippedTitle() == $title)
        {
            $request->setParameter('id', $post->getId());

            return $this->forward('post', 'show');
        }
    }

    $this->forward404();
}

```

*Listing
1-34*

The post list can call this permalink action instead of the show one for each post. In `sf_sandbox/apps/frontend/modules/post/templates/indexSuccess.php`, delete the `id` table header and cell, and change the `Title` cell from this:

```
<td><?php echo $blog_post->getTitle() ?></td>
```

*Listing
1-35*

to this, which uses a named rule we will create in a second:

```
<td><?php echo link_to($blog_post->getTitle(),
 '@post?title='.$blog_post->getStrippedTitle()) ?></td>
```

*Listing
1-36*

Just one more step: edit the `routing.yml` located in the `sf_sandbox/apps/frontend/config/` directory and add these rules at the top:

```

list_of_posts:
    url:    /latest_posts
    param: { module: post, action: index }

post:
    url:    /blog/:title
    param: { module: post, action: permalink }

```

*Listing
1-37*

Now navigate again in your application to see your new URLs in action. If you get an error, it may be because the routing cache needs to be cleared. To do that, type the following at the command line while in your `sf_sandbox` folder:

Listing 1-38 `$ php symfony cc`



Find more about smart URLs¹⁷.

Cleaning up the frontend

Well, if this is meant to be a blog, then it is perhaps a little strange that everybody is allowed to post! This isn't generally how blogs are meant to work, so let's clean up our templates a bit.

In the template `sf_sandbox/apps/frontend/modules/post/templates/showSuccess.php`, get rid of the 'edit' link by removing the line:

Listing 1-39

```
<a href="<?php echo url_for('post/edit?id='.$blog_post->getId())
?>">Edit</a>
&nbsp;
```

Do the same for the `sf_sandbox/apps/frontend/modules/post/templates/indexSuccess.php` template and remove:

Listing 1-40

```
<a href="<?php echo url_for('post/edit') ?>">New</a>
```

You should also remove the following methods from `sf_sandbox/apps/frontend/modules/post/actions/actions.class.php`:

- `executeEdit`
- `executeDelete`

This means that readers cannot post anymore, which is what is required.

Generation of the backend

To allow you to write posts, let's create a backend application by typing in the command line (still from the `sf_sandbox` project directory):

17. http://www.symfony-project.org/book/1_2/09-Links-and-the-Routing-System

```
$ php symfony generate:app backend
$ php symfony propel:init-admin backend post BlogPost
$ php symfony propel:init-admin backend comment BlogComment
```

Listing
1-41

This time, we use the admin generator¹⁸. It offers much more features and customization than the basic CRUD generator.

Just like you did for the frontend application, edit the layout (apps/backend/templates/layout.php) to add global navigation:

```
<div id="navigation">
  <ul style="list-style:none;">
    <li><?php echo link_to('Manage posts', 'post/index') ?></li>
    <li><?php echo link_to('Manage comments', 'comment/index') ?></li>
  </ul>
</div>
<div id="content">
  <?php echo $sf_data->getRaw('sf_content') ?>
</div>
```

Listing
1-42

You can access your new back-office application in the development environment by calling:

http://localhost/sf_sandbox/web/backend_dev.php/post

Listing
1-43

The screenshot shows a web browser window with the title 'Sf 1.1.4'. The browser's address bar shows 'http://localhost/sf_sandbox/web/backend_dev.php/post'. The page content includes two links: 'Manage posts' and 'Manage comments'. Below the links is a section titled 'post list' containing a table with the following data:

Id	Title	Excerpt	Body	Created at
1	Test post #1	This is an excerpt of my first post.	This is the detail, which tells you all about it.	19 October 2008 10:52
2	Test post #2	This is the excerpt for my second post.	More detail goes in the body of the post, for my second article.	19 October 2008 10:53

Below the table, it says '2 results' and there is a green 'create' button.

The great advantage of the generated admin is that you can easily customize it by editing a configuration file.

Change the `sf_sandbox/apps/backend/modules/post/config/generator.yml` to:

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  BlogPost
    theme:        default
  fields:
    title:        { name: Title }
    excerpt:      { name: Excerpt }
    body:         { name: Body }
    nb_comments:  { name: Comments }
    created_at:   { name: Creation date }
  list:
    title:        Post list
    layout:       tabular
    display:      [=title, excerpt, nb_comments, created_at]
    object_actions:
      _edit:      ~
```

Listing
1-44

18. http://www.symfony-project.org/book/1_2/14-Generators

```

        _delete:           ~
        max_per_page:      5
        filters:           [title, created_at]
    edit:
        title:             Post detail
        fields:
            title:         { type: input_tag, params: size=53 }
            excerpt:       { type: textarea_tag, params: size=50x2 }
            body:          { type: textarea_tag, params: size=50x10 }
            created_at:    { type: input_date_tag, params: rich=on }

```

Note that among the existing columns of the `blog_post` table, the admin will look for a column (or getter method) for `nb_comments`. Since this is a generated value and not a column, we can add a getter to our model (`sf_sandbox/lib/model/BlogPost.php`):

```

Listing 1-45 public function getNbComments()
{
    return count($this->getBlogComments());
}

```

Now refresh the Post administration screen to see the changes:

The screenshot shows the Symfony administration interface. At the top, there are navigation links for 'Manage posts' and 'Manage comments'. Below that is a 'Post list' table with the following data:

Title	Excerpt	Comments	Creation date	Actions
Test post #1	This is an excerpt of my first post.	0	19 October 2008 10:52	[edit] [delete]
Test post #2	This is the excerpt for my second post.	2	19 October 2008 10:53	[edit] [delete]

Below the table is a 'create' button. To the right of the table is a 'filters' sidebar with input fields for 'Title:' and 'Creation date:'. At the bottom of the sidebar are 'reset' and 'filter' buttons.

Restrict access to the backend

Currently the backend application can be accessed by everybody. We therefore need to add some access restrictions. In `apps/backend/config/`, edit the file called `security.yml` and reset the content to:

```

Listing 1-46 all:
    is_secure: on

```

Now you can no longer access these modules, unless you are logged in. However, currently the login action doesn't exist! Of course, we can easily add it. To do so, let's install a suitable plugin from the symfony website - `sfGuardPlugin`. Type the following at the command line:

```

Listing 1-47 $ php symfony plugin:install sfGuardPlugin

```

This will download the plugin from the symfony plugin repository. At this point the command line should give you an indication that the installation was successful:

```

Listing 1-48 $ php symfony plugin:install sfGuardPlugin
>> plugin    installing plugin "sfGuardPlugin"

```

```
>> sfPearFrontendPlugin Attempting to discover channel
"pear.symfony-project.com"...
>> sfPearFrontendPlugin downloading channel.xml ...
>> sfPearFrontendPlugin Starting to download channel.xml (663 bytes)
>> sfPearFrontendPlugin .
>> sfPearFrontendPlugin ...done: 663 bytes
>> sfPearFrontendPlugin Auto-discovered channel
"pear.symfony-project.com", alias
>> sfPearFrontendPlugin "symfony", adding to registry
>> sfPearFrontendPlugin Attempting to discover channel
>> sfPearFrontendPlugin "plugins.symfony-project.org"...
>> sfPearFrontendPlugin downloading channel.xml ...
>> sfPearFrontendPlugin Starting to download channel.xml (639 bytes)
>> sfPearFrontendPlugin ...done: 639 bytes
>> sfPearFrontendPlugin Auto-discovered channel
"plugins.symfony-project.org", alias
>> sfPearFrontendPlugin "symfony-plugins", adding to registry
>> sfPearFrontendPlugin downloading sfGuardPlugin-2.2.0.tgz ...
>> sfPearFrontendPlugin Starting to download sfGuardPlugin-2.2.0.tgz
(18,589 bytes)
>> sfPearFrontendPlugin ...done: 18,589 bytes
>> sfSymfonyPluginManager Installation successful for plugin
"sfGuardPlugin"
```

Next, the plugin needs to be enabled. Edit `sf_sandbox/apps/backend/config/settings.yml` to enable the login system, as follows. Uncomment the `all:` key and add the following:

```
# (Some stuff here)
all:
  .actions:
    login_module:    sfGuardAuth    # To be called when a non-authenticated
user
    login_action:    signin         # Tries to access a secure page

    secure_module:   sfGuardAuth    # To be called when a user doesn't have
    secure_action:   secure         # The credentials required for an action

  .settings:
    enabled_modules: [default, sfGuardAuth, sfGuardGroup,
sfGuardPermission, sfGuardUser]
```

*Listing
1-49*

Now, enable the new user system by editing `sf_sandbox/apps/backend/lib/myUser.class.php`, so that it reads:

```
class myUser extends sfGuardSecurityUser
{
}
```

*Listing
1-50*

Now, the model, forms and filters need to be built, the new SQL needs to be generated, and the database tables need to be updated:

```
$ php symfony propel:build-model
$ php symfony propel:build-forms
$ php symfony propel:build-filters
$ php symfony propel:build-sql
$ php symfony propel:insert-sql
```

*Listing
1-51*



When launching the `propel:insert-sql` task, symfony will drop all tables to re-create them. As during the development this happens a lot, symfony can store initial data in fixtures (see populating a database¹⁹ for more information).

Now, clear your cache again. Finally, create a user by running:

Listing
1-52

```
$ symfony guard:create-user --application=backend jon SoMePasSw0rD
```

Let's now make the post administration module the default one in our backend system. To do this, open up the `apps/backend/config/routing.yml` file and locate the `homepage` key. Change the module from `default` to `post`.

At that point, if you try to access the posts management (http://localhost/sf_sandbox/web/backend_dev.php/post), you will have to enter a valid username and password:

Find more about security²⁰.

Conclusion

Ok, the hour is out. You made it. Now you can use both applications in the production environment and play with them:

Listing
1-53

```
frontend: http://localhost/sf_sandbox/web/index.php/
backend:  http://localhost/sf_sandbox/web/backend.php/
```

At this point, if you meet an error, it might be because you changed the model after some actions were put in cache (cache isn't activated in the development environment). To clear the cache, simply type:

Listing
1-54

```
$ php symfony cc
```

See, the application is fast and runs smoothly. Pretty cool, isn't it? Feel free to explore the code, add new modules, and change the design of pages.

And don't forget to mention your working symfony applications in the symfony Wiki²¹!

19. http://www.symfony-project.org/book/1_2/16-Application-Management-Tools#Populating%20a%20Database

20. http://www.symfony-project.org/book/1_2/06-Inside-the-Controller-Layer#Action%20Security

21. <http://trac.symfony-project.org/wiki/ApplicationsDevelopedWithSymfony>