



What's new in symfony 1.2?

This PDF is brought to you by
SENSIOLABS 

License: Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License
Version: whats-new-1.2-en-2009-12-05

Table of Contents

Array

What's new in symfony 1.2?

This tutorial is a quick technical introduction for symfony 1.2. It is for developers who have already worked with symfony 1.0 and 1.1 and who want to quickly learn new features of symfony 1.2.

First, please note that symfony 1.2 is compatible with PHP 5.2.4 or later, whereas symfony 1.1 works with PHP 5.1 and symfony 1.0 with PHP 5.0.

If you want to upgrade from 1.1 or 1.0, please read the [UPGRADE¹](#) file found in the symfony distribution. You will have there all the information needed to safely upgrade your projects to symfony 1.2.

Propel

Propel has been upgraded to version 1.3, which replaces support for Creole with PDO, and includes many new features including: object instance pooling, master-slave connections, native nested-set support, and better date handling.

The `databases.yml` configuration file now uses the PDO syntax:

```
dev:
  propel:
    param:
      classname: DebugPDO

all:
  propel:
    class: sfPropelDatabase
    param:
      dsn:          mysql:dbname=example;host=localhost
      username:    username
      password:    password
      encoding:    utf8
      persistent:  true
      pooling:     true
      classname:   PropelPDO
```

*Listing
1-1*

The transaction api has change slightly: `->begin` has been renamed `->beginTransaction()` and `->rollback()` has been renamed `->rollBack()`. The `::doSelectRS` method has been renamed to `::doSelectStmt`.

For more information on Propel 1.3, please read the complete documentation² on their website.

1. http://www.symfony-project.org/installation/1_2/upgrade

2. <http://propel.phpdb.org/trac/wiki/Users/Documentation/1.3>

Doctrine

Doctrine is a first-class citizen in symfony 1.2. It means that symfony 1.2 comes bundled with both the Propel plugin and the Doctrine one. All the built-in features are available for both Propel and Doctrine, so feel free to choose the ORM that suits you better.

Request

You can now simulate PUT and DELETE requests from a browser by using the POST method and adding a special `sf_method` parameter:

```
Listing 1-2 <form action="#" method="POST">
  <input type="hidden" name="sf_method" value="PUT" />

  <!-- // ... -->
</form>
```

The `form_tag()` helper has been updated to automatically generate the hidden tag for methods different from GET or POST. So, the opening form tag from the code above can be generated by using the `form_tag()` helper like this:

```
Listing 1-3 <?php echo form_tag('#', array('method' => 'PUT')) ?>
```

As the `link_to()` helper now embeds a CSRF token if CSRF protection is enabled, you can check the token when the link is clicked by using the `checkCSRFProtection()` method:

```
Listing 1-4 public function executeDelete($request)
{
  $request->checkCSRFProtection();
}
```

The `checkCSRFProtection()` throws an exception if the token is not present or not valid.

Forms

Nested forms

One of the main limitation of Propel forms in symfony 1.1 was the inability to auto-save objects from embedded forms. This has been implemented in symfony 1.2, which means that when you save a Propel form, symfony will automatically save the main object and all the objects related to the embedded forms.

New `sfForm` methods

The `sfForm::renderFormTag()` method generates the opening form tag for the current form. It also adds the `enctype` attribute if the form needs to be multipart and adds a hidden tag if the form method is not GET or POST:

```
Listing 1-5 <?php echo $form->renderFormTag(url_for('@article_update'), array('method'
=> 'PUT')) ?>
```

The `sfFormPropel` class overrides `renderFormTag()` to automatically change the HTTP method based on the related object: if the object is new, the method will be `POST`, and if the object already exists, the method will be `PUT`.

The `sfForm::hasErrors()` method returns `true` if the form has some errors and `false` otherwise. This method returns `false` if the form is not bound. This method is quite useful in a template:

```
<?php if ($form->hasErrors()): ?>
    The form has some errors you need to fix.
<?php endif; ?>
```

*Listing
1-6*

The `sfForm::renderUsing()` method allows to render the form using a specific widget schema formatter. This allows to use a formatter directly within a template:

```
// in a template, the form will be rendered using the "list" form formatter
echo $form->renderUsing('list');
```

*Listing
1-7*

The `sfForm::renderHiddenFields()` method returns HTML for hidden widgets. This can be useful when a form's fields are rendered individually in a template:

```
<form action="<?php echo url_for('@some_route') ?>">
  <?php echo $form->renderHiddenFields() ?>
  <ul>
    <?php echo $form['name']->renderRow() ?>
  </ul>
  <input type="submit" />
</form>
```

*Listing
1-8*

The `sfForm::getStylesheets()` and `sfForm::getJavaScripts()` methods return the stylesheets and the JavaScripts files needed to render the form. These files can be defined in the form itself by overriding these two methods, or more commonly by each widget (see the `Widget` paragraph to learn how).

In a template, you can use the `include_javascripts_for_form()` and `include_stylesheets_for_form()` helpers to render them as HTML. They both take a form object as an argument:

```
<?php include_javascripts_for_form($form) ?>
<?php include_stylesheets_for_form($form) ?>
```

*Listing
1-9*

`sfForm` implements the `Iterator` and the `Countable` interfaces to allow easier iterating on form fields in a template:

```
<ul>
  <?php foreach ($form as $field): ?>
    <li><?php echo $field ?></li>
  <?php endforeach; ?>
</ul>
```

*Listing
1-10*

Cleaned-up values pre-processing

A way to pre-process the cleaned up values before they are used by Propel to update the object has been added.

If you want to pre-process a value, you need to add an `updateXXXColumn()` method to your form where `XXX` is the PHP name of the Propel column.

The method must return the processed value or `false` to remove the value from the array of cleaned up values:

```
Listing 1-11
class UserForm extends sfFormPropel
{
    // ...

    protected function updateProfilePhotoColumn($value)
    {
        // if the user has not submitted a profile_photo,
        // remove the value from the values as we want
        // to keep the old one
        if (!$value)
        {
            return false;
        }

        // remove the old photo
        // save the photo on the disk

        // change the value to the new file name
        return $filename;
    }
}
```

Validators

sfValidatorSchemaCompare

The `sfValidatorSchemaCompare` constant values have been changed. No change to your code need to be done, but now, you can use nice shortcuts. The following two examples are equivalent:

```
Listing 1-12
// symfony 1.1 and 1.2
$v = new sfValidatorSchemaCompare('left', sfValidatorSchemaCompare::EQUAL,
'right');

// symfony 1.2 only
$v = new sfValidatorSchemaCompare('left', '==', 'right');
```

sfValidatorChoice

`sfValidatorChoice` has now a `multiple` option to make it behave like a `sfValidatorChoiceMany`.

`sfValidatorPropelChoice` has now a `multiple` option to make it behave like a `sfValidatorPropelChoiceMany`.

sfValidatorDateRange

symfony 1.2 comes with a new `sfValidatorDateRange` validators to validate a range of dates.

sfValidatorFile

The `sfValidatedFile` class has been made a bit more flexible when it saves the file. Its constructor takes a new argument, the path to use when saving the file.

So, when saving a file, you can now:

- pass an absolute filename as before:

```
$file->save(sfConfig::get('sf_upload_dir').'/filename.pdf');
```

*Listing
1-13*

- pass a relative path (symfony will make it absolute by prepending the path):

```
$file->save('filename.pdf');
```

*Listing
1-14*

- pass null. The filename will be auto-generated by the `generateFilename()` method:

```
$file->save();
```

*Listing
1-15*

Moreover, the `save()` method returns the filename of the saved file (relative to the path argument).

As the `sfValidatedFile` object is created by the `sfValidatorFile` validator, a new `path` option has been added to the latter and is simply passed to the `sfValidatedFile` constructor:

```
$this->validatorSchema['file'] = new sfValidatorFile(array('path' =>
sfConfig::get('sf_upload_dir')));
```

*Listing
1-16*

The `sfFormPropel` takes advantages of these new enhancements to automate the saving of the files related to a Propel object, thanks to the new `saveUploadedFile()` method.

So, if you have a Propel object with a `file` column, and if you pass a path when defining the file validator in your form, symfony will take care of everything for you:

- if no file is uploaded in the form
 - do nothing and do not change anything in the database
- if a file is uploaded in the form
 - remove the old file
 - save the new one
 - set the file column to the filename (relative to the given path)

So, in the related action, you can now just use `$this->form->save()` to save the object and save the file automatically.

By default, symfony will generate a unique file name by computing a hash and adding the guessed extension. If you want to change the filename, you can simply create a `generateXXXFilename()` method in your object where `XXX` is the PHP name of the column. The method is given the `sfValidatedFile` object:

```
public function generateFileFilename(sfValidatedFile $file)
{
    return $this->getId().$file->getExtension($file->getOriginalExtension());
}
```

*Listing
1-17*

You can of course also create a `updateXXXColumn()` as we have seen before to override this behavior and manage the file uploading process by yourself.

`sfValidatorI18nChoiceCountry` and `sfValidatorI18nChoiceLanguage`

The `sfValidatorI18nChoiceCountry` and `sfValidatorI18nChoiceLanguage` validators had a required `culture` option in symfony 1.1. As the `culture` is not used in those validators, the `culture` option is now deprecated. It is still there to maintain backward compatibility but you don't need to provide it anymore.

Setting default values for required and invalid error codes related messages

Two new methods have been added to the `sfValidatorBase` class allowing to define default standard messages for required and invalid error codes:

```
Listing 1-18 sfValidatorBase::setRequiredMessage('This field is required');
sfValidatorBase::setInvalidMessage('The value provided for this field is
invalid');
```

Widgets

Widget options

All widgets have a new default option. This option sets the default value for the widget:

```
Listing 1-19 $widget = new sfWidgetFormInput(array('default' => 'default value'));
$widget->setDefault('default value');
echo $widget->getDefault();
```

You can also set default values for widget schemas:

```
Listing 1-20 $widget = new sfWidgetFormSchema(...);
$widget->setDefaults(array('name' => 'default value'));
$widget->setDefault('name', 'default value');
var_export($widget->getDefaults());
```

These defaults are taken into account in the forms when you don't override them with form defaults.

All widgets have a new `label` option. This option sets the label for the widget when used in a widget schema context:

```
Listing 1-21 $widget = new sfWidgetFormInput(array('label' => 'Enter your name here'));
$widget->setLabel('Enter your name here');
echo $widget->getLabel();
```

Widgets JavaScripts and stylesheets

Widgets can declare the JavaScripts and stylesheets they need to be rendered with the `getJavaScripts()` and `getStyleSheets()` methods:

```
Listing 1-22 class MyWidget extends sfWidgetForm
{
```

```

public function getJavaScripts()
{
    return array('/path/to/a/file.js');
}

public function getStylesheets()
{
    return array('/path/to/a/file.css' => 'all', '/path/to/a/file.css' =>
'screen,print');
}
}

```

The `getJavaScripts()` method must return an array of JavaScript files. The `getStylesheets()` method must return an array with files as keys and media as values.

New widgets

symfony 1.2 comes with some new widgets:

- `sfWidgetFormDateRange`
- `sfWidgetFormFilterInput`
- `sfWidgetFormFilterDate`
- `sfWidgetFormI18nSelectCurrency`
- `sfWidgetFormSelectCheckbox`

`sfWidgetFormChoice` family

There is also a new choice widget family:

- `sfWidgetFormChoice`
- `sfWidgetFormChoiceMany`
- `sfWidgetFormPropelChoice`
- `sfWidgetFormPropelChoiceMany`

By default, these widgets behave like their `Select` counterpart. But they are a wrapper on top of the `sfWidgetFormSelect`, `sfWidgetFormSelectRadio`, and `sfWidgetFormSelectCheckbox` widgets. They use one of these three widgets for the rendering. To change the default widget, you have some options:

- `expanded`:
 - if `false`, then the widget will be a `select` tag
 - if `true` and `multiple` is `false`, then the widget will be a list of `radio` tags
 - if `true` and `multiple` is `true`, then the widget will be a list of `checkbox` tags
- `renderer_options`: When creating the widget (`select`, `radio`, `checkbox`), this is the options you want to pass to the renderer widget
- `renderer_class`: The class to use instead of the default one
- `renderer`: You can also pass a widget object directly (this overrides the previous options).

Here are some example:

```

$widget = new sfWidgetFormPropelSelect(array('model' => 'Article'));

// is equivalent to
$widget = new sfWidgetFormPropelChoice(array('model' => 'Article'));

```

Listing
1-23

```
// change the rendering to use a radio list
$widget->setOption('expanded', true);

// create a multiple select
$widget = new sfWidgetFormPropelChoiceMany(array('model' => 'Article'));

// change the rendering to use a checkbox list
$widget->setOption('expanded', true);
```

These new widgets are now used by default for the Propel generated forms.

Response

`sfWebResponse::getCharset()`

The `sfWebResponse::getCharset()` returns the current response charset. This charset is automatically updated if you change it by setting the content type.

`sfWebResponse::getStylesheets()` and `sfWebResponse::getJavascrpts()`

The `getStylesheets()` and `getJavascrpts()` methods can now return all the stylesheets and JavaScripts ordered by position if you pass `sfWebResponse::ALL` as their first argument:

```
Listing 1-24 $response = new sfWebResponse(new sfEventDispatcher());
$response->addStylesheet('foo.css');
$response->addStylesheet('bar.css', 'first');

var_export($response->getStylesheets());

// outputs
array(
  'bar.css' => array(),
  'foo.css' => array(),
)
```

The `sfWebResponse::ALL` is also now the default value for the position argument.

Prototype and Scriptaculous

In symfony, 1.2 the bundled Prototype and Scriptaculous libraries and their associated helpers (`JavascriptHelper`) have been moved to a core plugin.

Core plugins behave like any plugin but are shipped with symfony. This will make the JavaScript and CSS files of the new `sfProtoculousPlugin` (the more or less unofficial name of the often featured duo) behave like real plugin assets. They are now in `web/sfProtoculousPlugin` rather than in `web/sf` (as it has been in 1.0 and 1.1). The `prototype_web_dir` setting will also now point to the new directory.

In addition some very basic javascript helpers that are reusable by any JS framework, have been extracted to a `JavascriptBaseHelper` which stays in core.

As a new addition, `javascript_tag()` now can behave as `slot()`. This allows such usage

```
<?php javascript_tag() ?>
alert('All is good')
<?php end_javascript_tag() ?>
```

Listing
1-25

Tests

```
sfBrowser::info()
```

When you write a lot of functional tests for a given module, it is sometimes useful to have some visual information about what it is being done. As of symfony 1.2, the `info()` method outputs some text to help categorize your tests:

```
$browser->
  info('First scenario')->
  // ... some tests
  info('Second scenario')->
  // ... some more tests
;
```

Listing
1-26

Debugging tests

When a problem occurs in a functional test, the HTML transferred to the browser help to diagnose the cause. As of symfony 1.2, this is quite easy to display the generated HTML without having to interrupt the fluent interface style:

```
$browser->
  get('/a_uri_with_an_error')->
  with('response')->debug()->
  // some tests that won't be executed
;
```

Listing
1-27

The `debug()` method will output the response headers and content and will interrupt the flow of the browser.

Testers

The `sfTestFunctionalBase` class now delegates the actual tests to `sfTester` classes. symfony 1.2 has several built-in tester classes:

- request: `sfTesterRequest`
- response: `sfTesterResponse`
- user: `sfTesterUser`
- view_cache: `sfTesterViewCache`

All the old methods from the test browser have been moved to one of the tester class:

<i>method name</i>	<i>tester class</i>	<i>new method name</i>
<code>isRequestParameter</code>	<code>sfTesterRequest</code>	<code>isParameter</code>
<code>isRequestFormat</code>	<code>sfTesterRequest</code>	<code>isFormat</code>
<code>isStatusCode</code>	<code>sfTesterResponse</code>	<code>isStatusCode</code>
<code>responseContains</code>	<code>sfTesterResponse</code>	<code>contains</code>
<code>isResponseHeader</code>	<code>sfTesterResponse</code>	<code>isHeader</code>

method name	tester class	new method name
checkResponseElement	sfTesterResponse	checkElement
isUserCulture	sfTesterUser	isCulture
isCached	sfTesterViewCache	isCached
isUriCached	sfTesterViewCache	isUriCached

The tester classes also comes with new test methods:

tester class	new method name
sfTesterRequest	hasCookie
sfTesterRequest	isCookie
sfTesterRequest	isMethod
sfTesterUser	isAuthenticated
sfTesterUser	hasCredential
sfTesterUser	isAttribute
sfTesterUser	isFlash

Here is how to use the new testers:

Listing
1-28

```
$browser->
  get('/')->
    with('request')->isParameter('module', 'foo')->
    with('request')->isParameter('module', 'bar')
;
```

The call to the with() method returns the tester object on which you can call any of its methods.

If you want to call several methods on the same tester object, you can put them in a 'block':

Listing
1-29

```
$browser->
  get('/')->

  with('request')->begin()->
    isParameter('module', 'foo')->
    isParameter('module', 'bar')->
    isMethod('get')->
    isFormat('html')->
  end()->

  with('response')->begin()->
    isStatusCode(200)->
    checkElement('body', 'foo')->
    isHeader('Content-Type', 'text/html; charset: UTF-8')->
    isRedirected(false)->
  end()
;
```

You can add your own tester by creating a class that inherits from sfTester and register it:

Listing
1-30

```
$browser->setTester('my_tester', 'myTesterClassName');
```

You can also extend the existing tester classes by overriding the default class names:

```
$browser->setTester('request', 'myTesterRequest');
```

*Listing
1-31*

To improve testing pages with forms, we added a `select()` and `deselect()` method to the test browser. This currently supports selecting and deselecting checkboxes and selecting radiobuttons, which will cause the other radiobuttons of the same name to be deselected.

```
$browser->
  select('aRadioId')->
  select('aCheckboxId')->
  deselect('anotherCheckbox');
```

*Listing
1-32*

Cookies

Cookies are now extensively supported in the `sfBrowser` and `sfTestBrowser` classes.

You can manage cookies between requests by using the `setCookie()`, `removeCookie()`, and `clearCookies()` methods of the `sfBrowser` class:

```
$b->
  setCookie('foo', 'bar')->
  removeCookie('bar')->
  get('/')->
  // ...

  clearCookies()->
  get('/')->
  // ...
```

*Listing
1-33*

You can also test cookies with the `hasCookie()` and `isCookie()` methods from the `sfTesterRequest` class:

```
$b->
  get('/')->
  with('request')->begin()->
    hasCookie('foo')->
    hasCookie('foobar', false)->
    isCookie('foo', 'bar')->
    isCookie('foo', '/a/')->
    isCookie('foo', '!z/')->
  end()
;
```

*Listing
1-34*

The `hasCookie()` method takes a Boolean as its second argument to be able to test the fact that a cookie is not set.

The second argument of the `isCookie()` method behaves as the second argument of the `checkElementResponse` method. It can be a string, a regular expression, or a negative regular expression.

The browser class also automatically expires cookies as per the `expire` value of each cookie.

Request

You can now test the HTTP method used by the request in your functional tests using the `isMethod()` method from the request tester:

```
Listing 1-35 $b->
    setField('login', 'johndoe')->
    click('Submit')->
    with('request')->isMethod('post');
```

Links

When you simulate a click on a button or on a link, you give the name to the `click()` method. But you don't have the possibility to differentiate two different links or buttons with the same name.

As of symfony 1.2, the `click()` method takes a third argument to pass some options.

You can pass a `position` option to change the link you want to click on:

```
Listing 1-36 $b->
    click('/', array(), array('position' => 1))->
    // ...
;
```

By default, symfony clicks on the first link it finds in the page.

You can also pass a `method` option to change the method of the link or the form you are clicking on:

```
Listing 1-37 $b->
    click('/delete', array(), array('method' => 'delete'))->
    // ...
;
```

This is very useful when a link is converted to a dynamic form generated with JavaScript. You can also simulate the CSRF token generated by the `link_to()` helper by passing a `_with_csrf` option:

```
Listing 1-38 $b->
    click('/delete', array(), array('method' => 'delete', '_with_csrf' =>
true))->
    // ...
    call('/delete', 'delete', array('_with_csrf' => true))->
    // ...
;
```

Forms

If you use the new form framework, you can now test the errors generated by the submitted form:

```
Listing 1-39 $browser->
    click('save', array(...))->
    with('form')->begin()->
    hasErrors()->
    isError('name', 'Required.')->>
    isError('name', '/Required/')->>
    isError('name', '!/Invalid/')->>
    isError('name')->
    isError('name', 1)->
    end()
;
```

You can also debug a form with the `debug()` method:

```
$browser->
  click('save', array(...))->
  with('form')->debug()->
  // some tests that won't be executed
;
```

*Listing
1-40*

It will output the submitted values and all the errors if any.

Propel

The Propel plugin comes with a propel tester. This tester is not registered by default:

```
$browser->setTester('propel', 'sfTesterPropel');
```

*Listing
1-41*

This tester provides a `check()` method to test a Propel model:

```
$browser->
  post('...', array(...))->
  with('propel')->begin()->
    check('Article', array(), 3)->
    check('Article', array('title' => 'new title'))->
    check('Article', array('title' => 'new title'))->
    check('Article', array('title' => 'new%'))->
    check('Article', array('title' => '!new%'), 2)->
    check('Article', array('title' => 'title'), false)->
    check('Article', array('title' => '!title'))->
  end()
;
```

*Listing
1-42*

It takes a model class name as its first argument, a `Criteria` object or an array of conditions as the second one. The third one can be one of:

- `true`: To check that the `Criteria` returns at least one object
- `false`: To check that the `Criteria` returns no object
- an integer to check the number of returned objects

Coverage

There is a new `test:coverage` task that output the code coverage for some given tests:

```
./symfony test:coverage test/unit/model/ArticleTest.php lib/model/
Article.php
```

*Listing
1-43*

The first argument is a test file or a test directory. The second one is the file or directory you want to cover.

If you want to know which lines are not covered, simply add the `--detailed` option:

```
./symfony test:coverage --detailed test/unit/model/ArticleTest.php lib/
model/Article.php
```

*Listing
1-44*

Loggers

symfony 1.2 comes with a new built-in logger: `sfVarLogger`. This logger class logs all the messages as an array for later use. It's up to you to get the logs and do something with them:

```
Listing 1-45
$log = new sfVarLogger();
$log->log('foo');

var_export($log->getLogs());

// outputs
array(
  0 => array(
    'priority'      => 6,
    'priority_name' => 'info',
    'time'          => 1219385295,
    'message'       => 'foo',
    'type'          => 'sfOther',
    'debug_stack'   => array()
  )
)
```

Each log is an associative array with the following keys: `priority`, `priority_name`, `time`, `message`, `type`, and `debug_stack`.

The `debug_stack` attribute is only set if you turns the `xdebug_logging` option to `true`. It then contains the stack trace returned by `xdebug` as an array.

The `sfVarLogger` is also the base class for the `sfWebDebugLogger` class. So, the `xdebugLogging` option of `sfWebDebugLogger` has been renamed to `xdebug_logging`.

Web Debug Toolbar

The web debug toolbar is now customizable. The toolbar is composed of panels. A panel is an object that extends `sfWebDebugPanel` and provides all the needed information to display if on the toolbar.

By default, the following panels are automatically registered:

<i>name</i>	<i>class name</i>
symfony_version	<code>sfWebDebugPanelSymfonyVersion</code>
cache	<code>sfWebDebugPanelCache</code>
config	<code>sfWebDebugPanelConfig</code>
logs	<code>sfWebDebugPanelLogs</code>
time	<code>sfWebDebugPanelTimer</code>
memory	<code>sfWebDebugPanelMemory</code>
db	<code>sfWebDebugPanelPropel</code>

You can customize the web debug toolbar by listening to the `debug.web.load_panels` event. The listener can then add new panels, remove existing ones, or even replace some.

The `sfWebDebugPanelLogs` panel filters the logs to be displayed by notifying the `debug.web.filter_logs` event. For example, the `sfWebDebugPanelPropel` and

`sfWebDebugPanelTimer` connect to this event to remove all Propel related logs and timer logs from the logs panel.

The total time displayed on the web debug toolbar is now computed with `$_SERVER['REQUEST_TIME']` (we used to compute it with `sfConfig::get('sf_time_start')`, which does not exist anymore). This means that the time displayed will be much larger than in symfony 1.0 and 1.1.

Tasks

The Propel tasks relying on Phing now output a clear error message if the embed Phing task fails.

Some CLI tasks takes an application name as an argument because they need to connect to the database. We need an application because the configuration can be customized on an application basis and all the symfony configuration system is based on the application level.

For these tasks, this argument has been removed in favor of an optional “application” option. If you don't provide the “application” option, symfony will take the database configuration from the project `databases.yml` file.

The following task signatures have been changed accordingly:

- `propel:build-all-load`
- `propel:data-dump`
- `propel:data-load`



This is possible because `sfDatabaseManager` now takes a project configuration or an application configuration. For the curious one, this works because `sfDatabaseConfigHandler` now returns a static or a dynamic configuration based on an array of configuration files (see the `execute()` and `evaluate()` methods).

To ease the debugging, the `propel:build-model`, `propel:build-all`, and `propel:build-all-load` tasks do not remove the generated XML schemas anymore if you pass the `--trace` option.

`propel:insert-sql`

The `propel:insert-sql` task removes all the data from the database. As it destroys information, it now asks the user to confirm the execution of the task. The same goes for the `propel:build-all` and `propel:build-all-load` tasks, as they call the `propel:insert-sql` task.

If you want to use these tasks in a batch and want to avoid the confirmation question, pass the `no-confirmation` option:

```
$ php symfony propel:insert-sql --no-confirmation
```

*Listing
1-46*

Before symfony 1.2, the `propel:insert-sql` task was the only task to read its database configuration information from `propel.ini`. As of symfony 1.2, it reads its information from `databases.yml`. So, if you use several different connections in your model, the task will take those into account. Thanks to this new feature, you can now use the `--connection` option if you want to only load SQL statements for a given connection:

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

*Listing
1-47*

You can also use the `--env` and the `--application` options to select a specific configuration to use:

```
Listing 1-48 $ php symfony propel:insert-sql --env=prod --application=frontend
```

New methods available for your tasks

The `sfTask` base class now provides three new methods to use in your tasks:

- `logBlock()`: Logs a message in a styled block (default styles are: INFO, COMMENT, QUESTION, and ERROR)
- `ask()`: Asks a question to the user and returns the given answer
- `askConfirmation()`: Asks a confirmation to the user and return true if the user confirmed, false otherwise

`propel:generate-module`

The `propel:generate-crud` has been renamed to `propel:generate-module`. The old task name is still available as an alias.

The `non-atomic-actions` option of `propel:generate-module` has been removed and some new options have been added:

- `singular`: The singular name for the actions and templates
- `plural`: The plural name for the actions and templates
- `route-prefix`: The route prefix to use
- `with-propel-route`: Whether the related routes are Propel aware

`propel:generate-module-for-route`

The new `propel:generate-module-for-route` generates a module based on a route definition:

```
Listing 1-49 php symfony propel:generate-module-for-route frontend articles
```

`app:routes`

As symfony now can auto-generate routes (see below), the `app:routes` task displays the list of current routes for an application:

```
Listing 1-50 php symfony app:routes frontend
```

If you want to get some details about a route, just add the route name:

```
Listing 1-51 php symfony app:routes frontend articles_update
```

`plugin:publish-assets`

As plugins are normally installed via a task and this invokes the creation of the symlink in the web directory this is required for core plugins as well.

To do this manually you need to invoke:

```
symfony plugin:publish-assets --only-core
```

The target is that the project creation and upgrade task do this for you.

Routing

Routing options

The routing takes two new options:

- `generate_shortest_url`: Whether to generate the shortest URL possible
- `extra_parameters_as_query_string`: Whether to generate extra parameters as a query string

By default, they are set to `false` in the default `factories.yml` configuration file to keep backward compatibility with symfony 1.0 and 1.1.

You can also enable or disable these options on a route basis:

```
articles:
  url:    /articles/:page
  param:  { module: article, action: list, page: 1 }
  options: { generate_shortest_url: true }
```

*Listing
1-52*

This route will generate the shortest URL possible. So, if you pass a page of 1, the generated URL will be `/articles`:

```
echo url_for('@articles?page=1');
// /articles
// would have been /articles/1 in symfony 1.1
```

*Listing
1-53*

```
echo url_for('@articles?page=2');
// /articles/2
```

Here is an example for `extra_parameters_as_query_string`:

```
articles:
  url:    /articles
  options: { extra_parameters_as_query_string: true }
```

*Listing
1-54*

This route will accept extra parameters and add them as a query string:

```
echo url_for('@articles?page=1');
// /articles?page=1
// would not have matched the route in symfony 1.1
```

*Listing
1-55*

```
echo url_for('@articles?page=2');
// /articles?page=2
```

sfRoute

The `sfPatternRouting` class now stores its routes as an array of `sfRoute` objects instead of a flat associative array.

By default, symfony uses the built-in `sfRoute` object, but you can change it by specifying a class entry in your `routing.yml` configuration file:

```
foo_bar:
  url:    /foo/:bar
  class: myRoute
  param: { module: foo, action: bar }
```

*Listing
1-56*

All the routing logic is contained in the `sfRoute` class, which means you can override the parsing and the generation logic with your own.



The `sfRoute` class is much more modular than the old `sfPatternRouting` class to allow easier customization of the default behavior. The “compilation” phase has been refactored into smaller methods, the code has been simplified, and it is based on a “real” tokenizer.

If you connect your routes with PHP code, you must now pass a `sfRoute` instance as the second argument for the `connect()`, `prependRoute()`, `appendRoute()`, and `insertRouteBefore()` methods:

```
Listing 1-57 $route = new myRoute('/foo/:bar', array('module' => 'foo', 'action' =>
'bar'));
$route->connect('foo_bar', $route);
```

When a request comes in, the matching route object is stored as an attribute of the request and you can get it from an action via the `getRoute()` method:

```
Listing 1-58 public function executeIndex()
{
    $route = $this->getRoute();
}
```

The `sfRoute` constructor takes an array of options as its last argument to allow the customization of each route. In the `routing.yml` configuration file, use to options key:

```
Listing 1-59 article:
    url:      /article/:id-:slug
    options: { segment_separators: [/, ., -] }
```

sfRequestRoute

Symfony has another built-in route, `sfRequestRoute`, which can enforce the HTTP method for your routes:

```
Listing 1-60 article:
    url:          /article/:id
    requirements: { sf_method: get }
    class:       sfRequestRoute
```

If you don't pass a `sf_method` requirement, symfony will enforces a `get`, or a `head` request.

This is made possible because the `parse()` method of `sfRouting` now takes a context as a second argument. When the request calls the routing, it passes the following context:

- `method`: The HTTP method name
- `format`: The request format
- `host`: The hostname
- `is_secure`: Whether the request was called with HTTPS or not
- `request_uri`: The full request URI
- `prefix`: The prefix to add to each generated route

With the previous routing configuration, the `article` route will only match requests with a `get` HTTP method. If you define several routes with the same `url` but different method requirements, you can pass `sf_method` as a parameter when you generate a route:

```
Listing 1-61 <?php echo link_to('Great article', '@article?id=1&sf_method=get') ?>
```

sfObjectRoute

symfony 1.2 comes with another route class that extends `sfRequestRoute`, `sfObjectRoute`.

`sfObjectRoute` binds a route to a PHP object. A `sfObjectRoute` will call some methods on your PHP class to get the object, or a collection of objects, related to the route.

```
article:
  url:    /article/:id
  class:  sfObjectRoute
  options: { model: Article, type: object, method: getById }
```

*Listing
1-62*

When an incoming URL matches the route, the `sfObjectRoute::getObject()` method will get the related object by calling the `Article::getById()` method.

The same goes for a collection of objects:

```
articles:
  url:    /articles/newest
  class:  sfObjectRoute
  options: { model: Article, type: list, method: getNewest }
```

*Listing
1-63*

sfPropelRoute

`sfPropelRoute` extends `sfObjectRoute` to bind a route to a Propel model.

In your actions, you can retrieve the related object or collection of objects by using the `sfObjectRoute::getObject()` and `sfObjectRoute::getObjects()` methods:

```
public function executeList($request)
{
  $this->articles = $this->getRoute()->getObjects();
}

public function executeShow($request)
{
  $this->article = $this->getRoute()->getObject();
}
```

*Listing
1-64*

Here is an example for an Article Propel object:

```
article:
  url:    /article/:id
  param:  { module: article, action: show }
  class:  sfPropelRoute
  options: { model: Article, type: object }
```

*Listing
1-65*

```
articles:
  url:    /articles
  param:  { module: article, action: list }
  class:  sfPropelRoute
  options: { model: Article, type: list, method:
getPublishedArticleCriteria }
```

If you don't define a method, `sfPropelRoute` will retrieve the object by building a Criteria object based on the available route variables.

The `sfPropelRoute` has two main advantages over `sfRoute`:

- When a request comes in and the route matches the URL, the matching `sfPropelRoute` object will be available in your actions, and the related `Article` object will be available by calling the `getObject()` method. Moreover, if the object does not exist in the database, it will automatically redirect the user to a 404 error page. This means less boiler-plate code in your actions.
- When you generate a link for this route, you can use the new `url_for()` signature and pass the article object directly for the parameters argument:

Listing 1-66 `<?php echo url_for('article', $article) ?>`

If you have to pass extra parameters (to enforce a HTTP method for example), you can use an array like this:

Listing 1-67 `<?php echo url_for('article', array('sf_subject' => $article, 'sf_method' => 'get')) ?>`

And of course, you can still use the full parameters:

Listing 1-68 `<?php echo url_for('article', array('id' => $article->getId(), 'slug' => $article->getSlug())) ?>`

Or use the internal URI:

Listing 1-69 `<?php echo url_for('@article?id='.$article->getId().'&slug='.$article->getSlug()) ?>`

The `sfPropelRoute` converts the article object to an array of parameters automatically.

`sfPropelRoute` does not only work with the primary key. It can also work with any column. In the following example, let's add the `slug` column to the route pattern:

Listing 1-70

```
article:
  url:      /article/:id/:slug
  param:    { module: article, action: show }
  class:    sfPropelRoute
  options: { model: Article, type: object }
```

But sometimes, you want to put in the pattern some information that does not exist in the database. In this case, you can pass a method option:

Listing 1-71

```
post:
  url:      /post/:id/:year/:month/:day/:slug
  param:    { module: article, action: show }
  class:    sfPropelRoute
  options: { model: Article, type: object, method: getObjectForRoute }
```

The `getObjectForRoute()` receives an array of parameters as its first argument and must return an `Article` object:

Listing 1-72

```
class ArticlePeer extends BaseArticlePeer
{
  static public function getObjectForRoute($parameters)
  {
    $criteria = new Criteria();
    $criteria->add(self::ID, $parameters['id']);
```

```

        return self::doSelectOne($criteria);
    }
}

```

`sfPropelRoute` also provide a `setListCriteria()` method to override the route parsed parameters. This is especially useful when you want to refine the objects returned by the route, based on some other parameters, like a filter for example:

```

public function executeList($request)
{
    $this->filters = new ArticleFormFilter();
    if ($request->isMethod('post'))
    {
        $this->filters->bind($request->getParameter('article_filters'));
        if ($this->filters->isValid())
        {
            $this->getRoute()->setListCriteria($this->filters->getCriteria());
        }
    }
}

$this->articles = $this->getRoute()->getObjects();
}

```

Listing
1-73

`sfRouteCollection`, `sfObjectRouteCollection`, and `sfPropelRouteCollection`

symfony also comes with “collection” route classes.

These classes (`sfRouteCollection`, `sfObjectRouteCollection`, `sfPropelRouteCollection`) generate standard routes based on a definition:

```

articles:
  class:  sfPropelRouteCollection
  options: { model: Article, module: article }

```

Listing
1-74

By default, this route will generate seven routes:

- list: articles GET /articles.:sf_format
- new: articles_new GET /articles/new.:sf_format
- create: articles_create POST /articles.:sf_format
- edit: articles_edit GET /articles/:id/edit.:sf_format
- update: articles_update PUT /articles/:id.:sf_format
- delete: articles_delete DELETE /articles/:id.:sf_format

You can customize the generation with several options:

- model: The model class name
- actions: The list of actions to generate (from the 7 listed above)
- module: The module name
- prefix_path: The prefix path to add for every route
- column: The column name for the primary key (id by default)
- with_show: Whether to add the show method or not
- segment_names: The segment names for new and edit
- model_methods: The methods to use to get a collection or an object
- requirements: The default requirements (by default, the id requirement is \d+)

- `route_class`: The route class to use (by default `sfObjectRoute` for `sfObjectRouteCollection` and `sfPropelRoute` for `sfPropelRouteCollection`)
- `collection_actions`: A list of actions to add as collection actions
- `object_actions`: A list of actions to add as object actions

Here is another example with some options:

```
Listing 1-75
articles:
  class:   sfPropelRouteCollection
  options:
    model:      Article
    module:     article
    prefix_path: /foo
    methods:    [ list, edit, update ]
    segment_names: { list: nouveau, edit: edition }
    collection_actions: { filter: post }
    object_actions: { publish: put }
```

URL helpers

`link_to()` and `url_for()`

If you want to create a link to a resource that must be submitted with the POST, PUT, or DELETE HTTP method, the `link_to()` helper can convert a link to a form if you pass the `method` option:

```
Listing 1-76
<?php echo link_to('@article_delete?id=1', array('method' => 'delete')) ?>
```

For POST, DELETE, and PUT methods, the `link_to()` helper also embeds a CSRF token if CSRF protection is enabled in `settings.yml`.

The old `post` option is still valid but deprecated:

```
Listing 1-77
// is deprecated
<?php echo link_to('@some_route', array('post' => true)) ?>

// and equivalent to
<?php echo link_to('@some_route', array('method' => 'post')) ?>
```

The `url_for()` and `link_to()` helpers support new signatures. Instead of an internal URI, they can now also take the route name and an array of parameters:

```
Listing 1-78
echo url_for('@article', array('id' => 1));
echo link_to('Link to article', '@article', array('id' => 1));
```

The old behavior still works without changing anything to your code.

Configuration

Before symfony 1.2, all the plugins installed under the `plugins` directory, and all built-in plugins were automatically loaded.

As of symfony 1.2, you need to enable the plugins you want to use in your projects. You can do this in your `ProjectConfiguration` class. Here is how to enable the Doctrine plugin and disable the Propel one:

```
public function setup()
{
    $this->enablePlugins('sfDoctrinePlugin');
    $this->disablePlugins('sfPropelPlugin');
}
```

*Listing
1-79*

You can add several plugins in one call by passing an array of plugin names:

```
public function setup()
{
    $this->enablePlugins(array('sfDoctrinePlugin', 'sfGuardPlugin'));
}
```

*Listing
1-80*

You can also change the order in which plugins are loaded by using the `setPlugins` method:

```
[php] public function setup() { $this->setPlugins(array('sfDoctrinePlugin',
'sfCompat10Plugin')); }
```

To enable the 1.0 compatibility plugin, you need to enable it in your configuration:

```
public function setup()
{
    $this->enablePlugins('sfCompat10Plugin');
}
```

*Listing
1-81*

By default, symfony only enables the Propel plugin.

You can also enable all installed plugins:

```
public function setup()
{
    $this->enableAllPluginsExcept('sfDoctrinePlugin');
}
```

*Listing
1-82*

The previous example allows you to enable all plugins except the Doctrine one. If you upgrade from 1.0 or 1.1, this line will make symfony behave like in symfony 1.0 and 1.1.

Plugin configuration

Plugins now have the option of providing a plugin configuration class. These plugin configuration classes setup autoloading for the plugin, and are instantiated in `sfProjectConfiguration`. This means symfony 1.2 tasks no longer require an application argument for plugin classes to be used. This allows plugins to connect to `command.*` events, something that was not previously possible.

Filters

symfony is now able to generate filters based on your model thanks to the `propel:build-filters` task:

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

*Listing
1-83*

This task generates filter form classes in `lib/filter/` by default.

To filter an Article model, here is the code you need in your actions:

```
Listing 1-84 $this->filters = new ArticleFormFilter();
if ($request->isMethod('post'))
{
    $this->filters->bind($request->getParameter('article_filters'));
    if ($this->filters->isValid())
    {
        $this->articles = ArticlePeer::doSelect($this->filters->getCriteria());
    }
}
```

And here is the related template code:

```
Listing 1-85 <form action="<?php echo url_for('@articles_filter') ?>" method="post">
<table>
    <?php echo $filters ?>
    <tr>
        <td colspan="2">
            &nbsp;&nbsp;&nbsp;<a href="<?php echo url_for('@articles') ?>">Reset</a>
            <input type="submit" value="Filter" />
        </td>
    </tr>
</table>
</form>
```

If you want to persist the filters between requests, here is a small working example:

```
Listing 1-86 public function executeList($request)
{
    $this->filters = new
ArticleFormFilter($this->getUser()->getAttribute('article_filters'));
    if ($request->isMethod('post'))
    {
        $this->filters->bind($request->getParameter('article_filters'));
        if ($this->filters->isValid())
        {
            $this->getUser()->setAttribute('article_filters',
$this->filters->getValues());
        }
    }

    $criteria =
$this->filters->buildCriteria($this->getUser()->getAttribute('article_filters'));
    $this->articles = ArticlePeer::doSelect($criteria);
}
```

Exception Templates

symfony now respects the current request format when rendering any uncaught exceptions. You can customize each format's output by adding a template to your project or application `config/error` directory.

For example, an uncaught exception during an XML request could render `config/error/exception.xml.php` when your application is in debug mode, or `config/error/error.xml.php` when your application is not in debug mode.

If you had customized the 500 error template in your project, you will need to manually move it to the new directory:

- For symfony 1.1: from `config/error500.php` to `config/error/error.html.php`
- For symfony 1.0: from `web/errors/error500.php` to `config/error/error.html.php`

Smaller improvements

symfony 1.2 also comes with a lot of improvements here and there. Here are some of them.

Action

In an action, you can now generate a URL by using the routing object directly by calling `generateUrl()`:

```
public function executeIndex()
{
    $this->redirect($this->generateUrl('homepage'));
}
```

*Listing
1-87*

The `generateUrl()` method takes a route name, an array of parameters, and whether to generate an absolute URL as its constructor arguments.

By default, when you use the `redirectIf()` or `redirectUnless()` methods in your actions, symfony automatically changes the response HTTP status code to 302.

These two methods now have an additional optional argument to change this default status code:

```
$this->redirectIf($condition, '@homepage', 301);
$this->redirectUnless($condition, '@homepage', 301);
```

*Listing
1-88*

The `redirect()` method already have this feature.

YAML

The YAML parser now supports full merge key (see <http://yaml.org/type/merge.html> for more examples).

```
$yaml = new sfYamlParser();

var_export($yaml->parse(<<<EOF
default_param: &default_param
  datasource: propel
  phptype:    mysql
  hostspec:  localhost
  database:  db
  username:  user
  password:  pass

param:
  <<: *default_param
  username:  myuser
  password:  mypass
```

*Listing
1-89*

```

EOF
));

// outputs
array(
  'default_param' => array(
    'datasource' => 'propel',
    'phptype' => 'mysql',
    'hostspect' => 'localhost',
    'database' => 'db',
    'username' => 'user',
    'password' => 'pass',
  )

  'param' => array(
    'datasource' => 'propel',
    'phptype' => 'mysql',
    'hostspect' => 'localhost',
    'database' => 'db',
    'username' => 'myuser',
    'password' => 'mypass',
  )
)

```

sfParameterHolder

The `has()` method of `sfParameterHolder` has been changed to be more semantically correct.

It now returns `true` even if the value is `null`:

Listing
1-90

```

$ph = new sfParameterHolder();
$ph->set('foo', 'bar');
$ph->set('bar', null);

$ph->has('foo') === true;
$ph->has('bar') === true; // returns false under symfony 1.0 or 1.1

```

The `sfParameterHolder::has()` method is used by the `hasParameter()` and `hasAttribute()` methods available for a large number of core classes.

image_tag() helper

In symfony 1.0 and 1.1 the `image_tag()` helper would generate the `alt` attribute of the `img`-tag from the filename. This now only happens if `sf_compat_10` is on. The new behaviour eases finding of unset alt attributes using a (x)html validator. As a bonus, there is now a `alt_title` option that will set alt and title attribute to the same value, which is useful for cross browser tooltips.

View Configuration

As of symfony 1.2, it is now possible to change the View class used by symfony to render the partials by setting the `partial_view_class` in `module.yml`. The class must extend `sfPartialView`. Similar to the `view_class` setting, symfony will automatically append `PartialView` to the `partial_view_class` value:

Listing
1-91

```
# module.yml
all:
  partial_view_class: my
```

Factories

The `sfViewCacheManager` class is now configurable in `factories.yml`:

```
view_cache_manager:
  class: sfViewCacheManager
```

*Listing
1-92*

View

You can override `sfViewCacheManager::generateCacheKey()` by defining a `sf_cache_namespace_callable` setting. As of symfony 1.2, the callable is now called with an additional argument, the view cache manager instance.

Events

symfony 1.2 comes with new events:

- `user.change_authentication`: notified when a user authentication status changes. The event takes the `authenticated` flag as an argument after the change occurred.
- `debug.web.load_panels` (cf. web debug toolbar paragraph)
- `debug.web.filter_logs` (cf. web debug toolbar paragraph)

I18N

Internally, the `sfCultureInfo` class is now only used as a singleton. Even if it is still possible to bypass the `getInstance()` method and instantiate a new object directly, it is deprecated. By using the singleton, the performance are better as we only instantiate one culture info object per request and it also means that you can now override some culture information globally in your configuration classes.

The `sfCultureInfo::getCountries()`, `sfCultureInfo::getCurrencies()`, and `sfCultureInfo::getLanguages()` methods now take an optional argument which allows to restrict the return value:

```
// will only return the FR and ES countries in english
$countries = sfCultureInfo::getInstance('en')->getCountries(array('FR', 'ES'));
```

*Listing
1-93*

The `sfCultureInfo::getCurrencies()` method now returns an array of currency names. In previous symfony versions, it returned an array with the symbol and the name. To get the old behavior, just pass `true` as the second argument:

```
$currencies = sfCultureInfo::getInstance('en')->getCurrencies(null, true);
```

*Listing
1-94*

To get the translation of a single country, language, or currency, you can now use the `getCountry()`, `getCurrency()`, and `getLanguage()` methods from the `sfCultureInfo` class.

The XLIFF support has been enhanced to allow the generated files to be edited by standard GUI tools (see <https://open-language-tools.dev.java.net/>).

Include Path Management

symfony 1.2 adds the static method `sfToolkit::addIncludePath()` for easily managing the PHP `include_path` setting. This method accepts two parameters: a single path or array of paths, and a position string (either "front" or "back"). The second parameter defaults to "front".

```
Listing // an example from sfPropelPlugin
1-95 sfToolkit::addIncludePath(array(
    sfConfig::get('sf_root_dir'),
    sfConfig::get('sf_symfony_lib_dir'),
    realpath(dirname(__FILE__).'../lib/vendor'),
));
```

Admin Generator

The admin generator has been rewritten for symfony 1.2. It is based on the form framework. To generate an admin module, use the `propel:generate-admin` task:

```
Listing $ php symfony propel:generate-admin backend Article
1-96
```

By default, the task adds a REST route for your module. You can also create a route by yourself and pass it as an argument instead of the model class name:

```
Listing $ php symfony propel:generate-admin backend article
1-97
```

The configuration is done via the `generator.yml`. All the configuration is now done under the `config` key:

```
Listing generator:
1-98 class: sfPropelGenerator
    param:
        model_class:      DemoCategory
        theme:            admin
        non_verbos_templates: true
        with_show:       false
        singular:        ~
        plural:          ~
        route_prefix:    categories
        with_propel_route: 1

    config:
        actions: ~
        fields: ~
        list: ~
        filter: ~
        form: ~
        edit: ~
        new: ~
```

You can now have different configuration for the `edit` and `new` form, and they both inherit from the `form` configuration.

The `name` entry has been renamed `label`.

The old admin generator has been kept for backward compatibility, so you can still use it if you want by running the `propel:init-admin` task.