symfony book

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About symfony

Symfony is an open-source web framework written in PHP5. Based on the best practices of web development, thoroughly tried on several active websites, symfony aims to speed up the creation and maintenance of web applications, and to replace the repetitive coding tasks by power, control and pleasure.

If you have been looking for a Rails/Django-like framework for PHP projects with features such as:

- simple templating and helpers
- cache management
- multiple environments support
- deployment management
- scaffolding
- smart URLs
- multilingual and I18N support
- object model and MVC separation
- Ajax support

...where all elements work seamlessly together, then symfony is made for you.

Its very small number of prerequisites make it easy to install on any configuration; you just need Unix or Windows with a web server and PHP 5 installed. It is compatible with almost every database system. In addition, it has a very small overhead, so the benefits of the framework don't come at the cost of an increase in hosting costs.

Using symfony is so natural and easy for people used to PHP and the design patterns of Internet applications that the learning curve is reduced to less than a day. The clean design and code readability will keep your delays short. Developers can apply agile development principles (such as DRY, KISS or the XP philosophy), focus on the application logic and avoid wasting time writing endless XML configuration files.

Symfony is aimed at building robust applications in an enterprise context. This means that you have full control over the configuration: from the directory structure to the foreign libraries, almost everything can be customized. To match your enterprise's development guidelines, symfony is bundled with additional tools to help you test, debug and document your project.

Last but not least, by choosing symfony you get the benefits of an active open-source community. It is entirely free and published under the MIT license.

Symfony is sponsored by Sensio, a French Web Agency well known for its innovative views on web development.

Symfony at a glance

Overview

Symfony is an object-oriented PHP5 framework based on the MVC model. Symfony allows for the separation of business rules, server logic and presentation views of a web application. It also contains numerous tools and classes aimed at shortening the development time of a complex web application.

What is a framework?

A framework adds new mechanisms on top of a programming language, and these mechanisms automate many of the development patterns used for a given purpose. A framework also adds structure to the code, and pushes the developer to write better, more readable and maintainable code. A framework also makes programming easier, since it packages complex operations into simple statements.

A framework is usually developed with the same language that it extends. A PHP5 framework is a set of files written in PHP5.

A framework will add layering to an application. In general, they divide applications in three layers:

- The **presentation logic** handles the interactions between the user and the software
- The **data source logic** carries the access to a database or other data providers
- The **domain logic**, or business logic, is the remaining piece. It involves calculation made on inputs, manipulation of data from the presentation, and dispatching of data source logic according to the commands received from the presentation.

Web application frameworks intend to facilitate the development of... web applications (websites, Intranets, etc.). Building a basic dynamic website can be easily achieved with existing programming languages, and PHP is known for its simplicity and broadly adopted for that purpose. With PHP alone, you can already query a database, manage session cookies, access files in the server, etc. But when it comes to building a more complex website, where business logic increases the volume of code to maintain, the need of a web application framework arises.

What is symfony?

Symfony is a complete framework designed to help and speedup the development of web applications.

It is based on the following concepts:

- compatible with as many environments as possible
- easy to install and configure
- simple to learn
- enterprise ready
- convention rather than configuration, supporting fallback calls
- simple in most cases, but still flexible enough to adapt to complex cases
- most common web features included

- compliant with most of the web "best practices" and web "design patterns"
- very readable code with easy maintenance
- open source

The main concept underlying the symfony framework is that the most common tasks are done automatically so that the developer can focus entirely on the specifics of an application. There is no need to reinvent the wheel every time a new web application is built.

To fulfill these requirements, symfony was written entirely in PHP5. It has been thoroughly tested in various real world projects, and is actually in use for high demand e-business websites. It is compatible with most of the available databases, among which:

- MySQL
- PostgreSQL
- Oracle
- MSSQL
- and any other database if a Creole driver exists for it

The symfony object model relies on three distinct layers:

- a database abstraction
- an object-relational mapping
- a Model-View-Controller model for the front and back-office

Common features of web projects are made easy since Symfony natively automates them:

- internationalization
- templating with helpers
- form validation
- cache management
- shopping cart management
- smart URLs
- scaffolding
- email sending
- Pagination
- AJAX interactions

In addition, to fulfill the requirements of enterprises having their own coding guidelines and project management rules, symfony can be entirely reconfigured using YAML configuration files. It provides by default, several development environments, and is bundled with tools to easily achieve the following operations:

- prototyping
- content management
- live configuration changes
- deployment
- unit testing
- applicative testing
- logging

- debugging

Symfony uses some code fragments of other open source projects:

- Creole, for the database abstraction layer
- Propel, for the object-relational mapping layer
- Mojavi, for the Model-View-Controller model layer

Is symfony made for me?

Whether you are a PHP5 expert or a beginner in web application programming doesn't matter. The main factor that should direct your decision is the size of your project.

If you want to develop a simple website with five to ten pages, having limited access to a database and no obligation for performance, availability or documentation, then you should stick with PHP alone. You wouldn't take much advantage of the features of a web application framework, and using object orientation or a MVC model would only slow down your development. Symfony is not optimized to run efficiently on a shared server with only CGI support.

On the other hand, if you develop bigger web applications, with a heavy business logic, PHP alone is not enough. If you plan to maintain or extend your application in the future, you will need your code to be lightweight, readable and effective. If you want to use the latest advances in user interaction (like AJAX) in an intuitive way, you can't just write hundreds of lines of JavaScript. If you want to have fun and develop fast, then PHP alone will probably be disappointing. For all these cases, symfony is made for you.

And of course, if you are a professional web developer, you already know all the benefits of web application frameworks, and you need one which is mature, well documented, and with a large community. Search no more, for symfony is your solution.

Installing symfony

Overview

This chapter describes the steps to install the symfony framework, either from a 'sandbox' archive, from a PEAR package or manually from the source repository.

Introduction

The symfony framework is a set of files written in PHP. A project based on symfony uses these files, so installing symfony means getting these files and making them available for your project.

Symfony can be used for one or many projects. If you work on a single project, you may want to embed the framework within the application you develop. On the other hand, if you choose to use symfony for more than one project, you will prefer to keep all the symfony files in one single place to make the upgrade easier.

In addition, when developing an application, you will probably need to install symfony twice: Once for your development environment, and once for the host server (unless your hosts already has symfony installed).

All these different needs have different answers, that's why you have several alternatives to install symfony:

- The **sandbox** is an empty symfony project where all the required libraries are already included, and where the basic configuration is already done. It is made mostly for symfony beginners, who want to play with the framework or try the tutorials without installing anything.
- The **PEAR installation** is recommended to those who want to run several symfony-based projects, with an easy way to upgrade. It requires PEAR version 1.4.0 or higher, which is bundled in most PHP distributions.
- The **manual installation** is meant to be used only by advanced PHP developers, who want to take advantage of the latest patches or add features of their own.

Note that the symfony framework evolves quickly, and that a new stable version can come out only a few days after your first installation. You have to think of the framework upgrade as a major concern, and that's why the first solution is not recommended if you develop a website for real. In fact, an application developed with the sandbox can not be upgraded easily.

Prerequisites

In order to use symfony, you need to have PHP5 and a web server installed. If you want to use the PEAR install, you will also need PEAR. If you plan on using persistent data, a database will be needed. Symfony is compatible with MySQL, PostgreSQL, Oracle, MSSQL, and any other database having a Creole driver.

> **Note**: You can find many LAMP packages offered for download, which contain the Apache server, the MySQL database and PHP all bundled together and configured for a given platform (XAMPP for many platforms, MAMP for Macintosh, WAMP for Windows, etc.). If you don't have PHP5 installed, they are probably the best option for you to start.

The standard distributions of PHP5 normally work perfectly with symfony. However, as all the installations are different, and because many of the developers compile PHP themselves, it sometimes happens that your web server throws errors because of missing components in PHP. If you can't manage to get symfony working, check out the installation forum, where many individual cases were solved and can help to solve yours.

> **Note**: Some packages sometimes embed both PHP4 and PHP5. Symfony only works with PHP5. To check the version of PHP that you are using, type in a command line:
>
> ```
> $ php -v
> ```

The sandbox

The sandbox is a simple archive of files. It contains an empty symfony project where all the required libraries (symfony, pake, creole, propel and phing) are already included. To install it, just unpack the archive under the root web directory configured for your server (usually `web/`). It will work "out of the box", without configuration or any additional package.

The sandbox is intended for you to practice with symfony in a local computer, not really to develop complex applications that may end up on the web. However, the version of symfony shipped with the sandbox is fully functional and equivalent to the one you can install via PEAR. Beware that the sandbox is not easily upgradeable.

Get the sandbox here: http://www.symfony-project.com/get/sf_sandbox.tgz. After unpacking the archive, test the sandbox by requesting the following URL:

```
http://localhost/sf_sandbox/web/
```

You should see a congratulations page.

f you are in the `sf_sandbox/` directory, you can use the command line to do usual site management operations. For instance, to clear the cache, type:

```
$ ./symfony.sh clear-cache                    (*nix)
symfony clear-cache                           (Windows)
```

To discover all the available actions of the symfony command line, type:

```
$ ./symfony.sh -T                             (*nix)
symfony -T
```

Refer to the included README file for more information.

After downloading the sandbox, you might want to follow the My first project tutorial to discover the basics of symfony development.

Installing symfony from a PEAR Package

PEAR setup

PEAR is used by PHP to install libraries from a central repository. The symfony project has its own repository, or channel. Note that channels are only available since version 1.4.0 of PEAR, so you should upgrade if your version is older:

```
$ pear upgrade PEAR
```

The first thing to do is to add the 'symfony' channel:

```
$ pear channel-discover pear.symfony-project.com
```

To see the libraries available in this channel:

```
$ pear remote-list -c symfony
```

Symfony installation

Now you are ready to install the latest stable version of symfony and all its dependencies with:

```
$ pear install symfony/symfony
```

That's it: symfony is installed. You may now create a new project with the new command line tool `symfony` and use the classes and methods of the libraries.

Symfony needs a few other packages to run; some are included in the installation, and some require you to install them if they are not present:

- pake: installed automatically. You will learn more about the Pake utility in the next chapter.
- creole: installed automatically (used for database access)
- propel: installed automatically (used for object/relational mapping)
- phing: requires manual installation

  $ pear install http://phing.info/pear/phing-current.tgz

To have a glimpse of all the tasks that you can perform directly with the command-line tool, type:

```
$ symfony -T
```

Where are the symfony files ?

The symfony libraries are now installed in:

```
$php_dir/symfony/      main libraries
$data_dir/symfony/     skeleton of symfony applications, default modules and configuration
$doc_dir/symfony/      documentation
$test_dir/symfony/     unit tests
```

The `_dir` variables are part of your PEAR configuration. To see their value, type:

```
$ pear config-show
```

As a matter of fact, the exact location of files is not very important, since the installation provides you with a new executable, `symfony`, which will do all the work for you (project creation, application installation and initialization, etc.). Only the `$doc_dir` directory will be needed in your web server configuration, since it contains some default files (stylesheets and images) that can be required by some pages of symfony projects. This configuration will be detailed in the next chapter.

Just as a reminder, the default locations for Unix and Windows are as follow:

Unix	Windows	-
usr/local/lib/php/symfony/	c:/Program Files/php/pear/symfony/	main executable, main libraries
usr/local/lib/php/data/symfony/	c:/Program Files/php/pear/data/symfony/	skeleton of symfony applications
usr/local/lib/php/doc/symfony/	c:/Program Files/php/pear/doc/symfony/	documentation
usr/local/lib/php/test/symfony/	c:/Program Files/php/pear/test/symfony/	unit tests

Getting nightly builds

The process described above will install the latest stable version. In order to have the latest bug corrections between two stable versions, you may wish to install the latest nightly build. These builds are published in the symfony channel with a 'beta' tag, and you can install them with:

```
$ pear install symfony/symfony-beta
```

If you already have a beta version installed, get the latest build with:

```
$ pear upgrade symfony/symfony-beta
```

> **Note**: in some Windows platforms, it appears that the PEAR utility uses its own web cache and doesn't actually require the symfony channel to check the latest version. Make sure you empty your PEAR cache before you ask for an upgrade (use `pear config-show` to get the location of the cache folder).

When upgrading your symfony installation, don't forget to clear the cache of all the applications using it:

```
$ cd myproject
$ symfony clear-cache
```

If you want to be be more selective in clearing the cache (to keep any existing HTML cache and clear only the configuration of each application), you'd rather use:

```
$ cd myproject
$ symfony clear-cache myapp1 config
$ symfony clear-cache myapp2 config
```

You can check which version of symfony is installed by typing:

```
$ symfony -V
```

Installing symfony by hand

If you don't want to use PEAR, you can still download the latest version directly from the SVN repository, by requesting a checkout, and install it by hand:

```
$ mkdir /home/steve/mysymfony
$ cd /home/steve/mysymfony
$ svn co http://svn.symfony-project.com/trunk/ .
```

For your project to make use of this installation, you will have to create two symbolic links in you project from the `lib` directory to the symfony `lib` directory:

```
$ cd /home/steve/myproject
$ ln -sf /home/steve/mysymfony/lib lib/symfony
$ ln -sf /home/steve/mysymfony/data data/symfony
```

In addition, you will have to make the shortcuts to allow the call to the command line tools `pake` and `symfony`.

> **Note**: You must also install Phing and Pake. Just install them under the `lib/symfony` directory.

Installing symfony without command line

If you want to host your website in a server that you can't access with a command line (and that might happen if your hosting provider only allows FTP access), you will need to install the two symfony directories manually in your project tree structure.

First, get the .tgz file of the symfony framework package from the symfony project website. Unpack it into a temporary folder. You will see the following file structure:

```
package.xml
symfony/
  LICENSE
  bin/
  data/
  lib/
```

The `bin` directory will not be of any use since you can't call a command.

Let's suppose that you already transferred your symfony-enabled project to a distant server. The distant tree structure will look like:

```
myproject/
  apps/
    myapp/
  batch/
```

```
cache/
config/
data/
doc/
lib/
log/
test/
web/
```

To learn more about the file structure of symfony projects, read the related chapter.

Now you need to copy some files from the symfony package into your project directory. First, create a `symfony` subdirectory in the `data` and `lib` directories of your distant project. Second, transfer the files as follows:

-	**local**	-	**distant**
copy	symfony/data/*	to	myproject/data/symfony/
copy	symfony/lib/*	to	myproject/lib/symfony/

If you're unsure of what to expect, you can download the sandbox to see how a project embedded with the framework should look like. In addition, the sandox creation script details all the operations needed to built a project able to run on its own.

> **Hint**: some hosting providers won't run scripts in php5 unless their suffix is `.php5`. If this is the case of your distant server, you will need to rename the `index.php` file found in `myproject/web/` into `index.php5`. Don't bother to rename the other files : since symfony uses a front controller, only the `index.php` file will have to be declared as a php5 script (learn more about the front controller feature in the controller chapter)

User-contributed guides

The symfony wiki contains a few articles describing step-by-step installations on various platforms.

The forum has a dedicated section where many installation issues have already been solved, and where you will find a symfony user willing to help you in minutes if you have a new problem.

Project creation and the 'symfony' command

Overview

This chapter describes the logical structure of a Symfony project, and details the use of the *symfony* command to initiate your project structure.

Introduction

In symfony, a **project** is a set of services and operations available under a given domain name, sharing the same object model.

Inside a project, the operations are grouped logically into **applications**; an application can normally run independently of the other applications of the same project.

In most cases, a project will contain two applications, one for the front-office, and one for the back-office, sharing the same database. But you can also have one project containing lots of mini-sites, each site being a different application. Note that hyperlinks between applications have to be in the absolute form.

Each application is a set of one or more **modules**, each module bringing a particular feature. A module usually represents a page or a group of pages with a similar purpose. Example modules could be `home`, `articles`, `help`, `shoppingCart`, `account`, etc.

Modules contain **actions**. They represent the various actions that can be done in a module, for instance a `shoppingCart` module can have `add`, `show` and `update` actions. Dealing with actions is almost like dealing with pages in a classic web application.

If this represents too many levels for a beginning project, it is very easy to group all actions into one module, so that the file structure can be kept simple. When the application gets more complex, it will be time to organize actions into logical modules.

An application can run in various **environments** to use, for instance, different configurations or databases. By default, every new application can run in three environments (development, test and production). An application can have as many environments as needed. The difference between environments is the configuration.

For instance, a test environment will log alerts and errors, while a production environment will only log errors. The cache acceleration is often deactivated in a development environment but activated in test and production environments. The development and test environments may need test data, stored in a database distinct from the production one. All environments can live together on the same machine, although a production server generally contains only the production environment.

> **Note**: If you use symfony from a sandbox, you don't need to setup a project nor an application, since the sandbox already has one 'sf_sandbox' project and one 'frontend' application. You don't need to setup the web server either, since your application is in the `web/` root directory.

Pake

SymFony uses a dedicated tool called Pake to administer projects, applications, and modules. Pake is a php tool similar to the Rake command, a Ruby translation of the `make` command. It automates some administration tasks according to a specific configuration file called `pakefile.php`. But since you use the `pake` tool just by typing `symfony` in a command line, everything becomes much simpler than it sounds.

To get the full list of the available administration operations, simply type from your project directory:

```
$ symfony -T
```

The description of the CLI tasks used during the early stage of a project follows. A full reference of the CLI tasks is available in the CLI chapter.

Project setup

First of all, you must create the directory that will contain all your project files:

```
$ mkdir /home/steve/myproject
```

Then, in order to initialize the project and generate the basic files and directories necessary for runtime, simply type:

```
$ cd /home/steve/myproject
$ symfony init-project myproject
```

Here is an overview of the file structure created:

```
apps/
batch/
cache/
config/
data/
doc/
lib/
log/
test/
web/
```

> **Note**: If you have specific requirements for your project, symfony can adapt to any custom file structure. All the paths in the symfony scripts use a set of special parameters defined in a file called `constants.php`, which can be customized as described in the file structure chapter.

The `symfony` command must always be called from a project root directory (`myproject/` in the example above), because all the tasks performed by this command are project specific. If called from one of the project subdirectories, the command will raise an error.

Application Setup

The project is not yet ready to be seen; it requires at least one application. To initialize it, use the `symfony init-app` command and pass the name of the application as an argument:

```
$ symfony init-app myapp
```

This will create a `myapp` directory in the `apps/` folder of the project root, with a default application configuration and a set of directories ready to host the file of your website:

```
apps/
  myapp/
    config/
    i18n/
    lib/
    modules/
    templates/
```

Some php files corresponding to the front controllers of each default environment are also created in the project root `web` directory:

```
web/
  index.php
  myapp_dev.php
```

`index.php` is the *production* front controller of the new application. Because you created the first application of the project, symfony created a file called `index.php` instead of `myapp.php` (if you now add a new application called `mynewapp`, the new production front controller will be named `mynewapp.php`). To run your application in the *development* environment, call the front controller `myapp_dev.php`.

> **Note**: If you carefully read the introduction, you may wonder where the `myapp_test.php` file is located. As a matter of fact, the *test* environment is used to do unit testing of your applications components and doesn't require a front controller. Refer to the unit testing chapter to learn more about it.

From now on, the `/home/steve/myproject/` directory will be considered as the root of your project. The root path is stored in a constant called `SF_ROOT_DIR`, defined in the file `index.php`, and we will use this name instead of the real path to avoid confusing the readers who aren't named Steve.

Web server setup

To be able to access and test the new application, the web server has to be configured. Here is an example for Apache, where a new VirtualHost is added in the `httpd.conf` file:

```
<Directory "/$data_dir/symfony/web/sf">
  AllowOverride All
  Allow from All
</Directory>
<VirtualHost *:80>
  ServerName myapp.example.com
  DocumentRoot "/home/steve/myproject/web"
```

```
  DirectoryIndex index.php
  Alias /sf /$data_dir/symfony/web/sf

  <Directory "/home/steve/myproject/web">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

> **Note**: In the above configuration, the `$data_dir` placeholders have to be replaced by your PEAR data directory. For example, for *nix, you should type:
>
> ```
> Alias /sf /usr/local/lib/php/data/symfony/web/sf
> ```
>
> You will find more about the PEAR directories in the installation chapter.

Restart Apache, and that's it: the newly created application can now be called and viewed through a standard web browser at the url:

```
http://myapp.example.com/index.php/
```

or, in debug mode:

```
http://myapp.example.com/myapp_dev.php/
```

> **Note**: Symfony uses the `mod_rewrite` module to display 'smart' urls. If your version of apache is not compiled with `mod_rewrite`, check that you have the mod_rewrite DSO installed and the following lines in your `httpd.conf`:
>
> ```
> AddModule mod_rewrite.c
> LoadModule rewrite_module modules/mod_rewrite.so
> ```
>
> You will learn more about the smart urls in the routing chapter.

Symfony is compatible with other server configurations. You can, for instance, access a symfony application using an alias instead of a virtual host. To discover more about web server configuration, refer to the related chapter.

Module setup

Your new application is not very impressive; it dramatically lacks functionalities. If you want functionalities, you need a module to put them into. Once again, the `symfony` command will be used for that, with the parameters `init-module`, the application name and the new module name:

```
$ symfony init-module myapp mymodule
```

The created tree structure is:

```
modules/
  mymodule/
    actions/
```

```
    config/
    lib/
    templates/
    validate/
```

The new module is ready to be used:

```
http://myapp.example.com/index.php/mymodule
```

Since you need to start getting to work right away, edit the file `myapp/modules/mymodule/templates/indexSuccess.php` and type in:

```
Hello, world !
```

Save it, refresh the page in the browser and voilÃ !

Source versioning

Once the setup of the application is done, it is recommended to start a source versioning process. Symfony natively supports CVS, although Subversion is recommended. The following examples will show the commands for Subversion, and take for granted that you already have a Subversion server installed and that you wish to create a new repository for your project. For Windows users, a recommended Subversion client is TortoiseSVN. For more information about source versioning and the commands used here, please consult the Subversion documentation.

The example below assumes that $SVNREP_DIR is defined as an environment variable. If you don't have it defined, you will need to substitute the actual location of the repository in place of $SVNREP_DIR.

So let's **create** the new repository for the `myproject` project:

```
$ svnadmin create $SVNREP_DIR/myproject
```

Then the base structure (layout) of the repository is created with the `trunk`, `tags` and `branches` directories with this pretty long command:

```
$ svn mkdir -m "layout creation" file:///$SVNREP_DIR/myproject/trunk file:///$SVNREP_DIR/myprojec
```

This will be your first revision. Now you have to **import** the files of the project except the cache and log temporary files:

```
$ cd /home/steve/myproject
$ rm -rf cache/*
$ rm -rf log/*
$ svn import -m "initial import" . file:///$SVNREP_DIR/myproject/trunk
```

Check the committed files by typing

```
$ svn ls file:///$SVNREP_DIR/myproject/trunk/
```

That seems good. Now the SVN repository has the reference version (and the history) of all your project files.

This means that the files of the actual `/home/steve/myproject` directory need to refer to the repository. To do that, first rename the `myproject` directory - you will erase it soon if everything works well - and do a **checkout** of the repository in a new directory:

```
$ cd /home/steve
$ mv myproject myproject.origin
$ svn co file:///$SVNREP_DIR/myproject/trunk myproject
$ ls myproject
```

That's it. Now you can work on the files located in `/home/steve/myproject/` and **commit** your modifications to the repository. Don't forget to do some cleanup and erase the `myproject.origin` directory, which is now useless.

There is one remaining thing to setup. If you commit your working directory to the repository, you may copy some unwanted files, like the ones located in the `cache` and `log` directories of your project. So you need to specify an ignore list to svn for this project. You also need to set full access to the `cache/` and `log/` directories again - SVN doesn't store the access rights:

```
$ cd /home/steve/myproject
$ svn propedit svn:ignore .
$ chmod 777 cache
$ chmod 777 log
```

The default text editor configured for SVN should launch. If this doesn't happen, make subversion use your preferred editor by typing

```
$ export SVN_EDITOR=<name of editor>
$ svn propedit svn:ignore .
```

Now simply add the subdirectories of `myproject` that SVN should ignore when committing:

```
cache
log
```

Save and quit. You're done.

File structure explained

Overview

The tree structure of a symfony project is consistent, although it can be completely customized. This chapter will help you to get used to this structure and to understand the logic behind it.

Introduction

All web projects generally share the same architectural needs:

- database
- static files (HTML, images, javascripts, stylesheets, etc.)
- files uploaded by the site users and administrators
- PHP classes and libraries
- foreign libraries
- batch files
- log files
- configuration files
- etc.

To allow developers to adapt to any existing symfony project, it is recommended to follow the default tree structure described below. This also speeds up project creation, since the default tree structure is automatically created when initializing every project, application or module.

Root tree structure

These are the directories found at the root of a ymfony project:

```
apps/
  fo/
  bo/
batch/
cache/
config/
data/
  sql/
doc/
  api/
lib/
  model/
log/
test/
web/
  css/
  images/
  js/
  uploads/
```

The `batch` directory is used for php files called from a command line or a scheduler to run batch processes.

The cache of the project, used to speed up the answer to web requests, is located in the `cache` directory. Each application will have a subdirectory in there, containing preprocessed HTML and configuration files.

The general configuration of the project is stored in the `config` directory.

In the `data` directory, you can store the data files of the project, like a database schema, a SQL file that creates tables, or even a SQLite database itself if you need.

The `doc` directory stores the project documentation, including your own documents and the documentation generated by phpdoc (in the `api` subdirectory).

The `lib` directory is dedicated to foreign classes or libraries. Here you can add the code that needs to be shared among your applications. The `model` subdirectory stores the object model of the project.

The `log` directory stores the applicable log files generated directly by symfony. It can also contain web server log files, database log files or log files from any part of the project. There is normally one file per application and per environment (ex: `myapp_prod.log`)

The `test` directory contains unit tests written in PHP and compatible with the SimpleTest testing framework. During the project setup, symfony automatically adds some stubs with a few basic tests.

The `web` directory is the root for the web server. The only files accessible from the Internet are the ones located in this directory. It will be described in detail in a few moments.

Last but not least, the `apps` directory contains one directory for each application of the project (typically `fo` and `bo` for the front and back office, `myapp` in the example).

Application tree structure

The tree structure of all application directories is the same:

```
config/
i18n/
lib/
modules/
templates/
  layout.php
  error.php
  error.txt
```

The `config` directory holds a hefty set of YAML configuration files. This is where most of the application configuration is, apart from the default parameters that can be found in the framework itself. Note that the default parameters can still be overridden here if needed. You can find more about application configuration in the next chapter.

The `i18n` directory contains files used for the internationalization of the application. These files can be in the XLIFF or in the GetText format. You can even bypass this directory if you choose to use a database for your internationalization.

The `lib` directory contains classes and libraries that are specific to the application.

The `modules` directory stores all the modules that contain the features of the application.

The `templates` directory lists the global templates of the application, the ones that are shared by all modules. By default, it contains a `layout.php` file, which is the main layout in which the module templates are inserted; an `error.php` file, used to output errors on a web request; and a `error.txt` file used to output errors when the application is called without a web browser (for instance during unit tests). Other global templates are to be added here - if your application uses popups sharing the same layout, you could add a `popuplayout.php` for example.

The directories `i18n`, `lib` and `modules` are empty for a new application.

The directory name of an application is to be determined during its setup. This name is accessible via the `sfConfig` object by referencing `sfConfig::get('sf_app_dir')`.

The classes of an application are not able to access methods or attributes in other applications of the same project. Also note that hyperlinks between two applications of the same project must be in absolute form.

Module tree structure

Each application contains one or more modules. Each module has its own subdirectory in the `modules` directory, and the name of this directory is chosen during the setup.

This is the typical tree structure of a module:

```
actions/
  actions.class.php
config/
lib/
templates/
  indexSuccess.php
validate/
```

The `actions` directory generally contains a single class named `actions.class.php`, in which you can store all the actions of the module. Different actions of a module can also be written in separate files.

The `config` directory can contain custom configuration files with local parameters for the module.

The `lib` directory contains classes and libraries specific to the module.

The `templates` directory contains the templates corresponding to the actions of the module. A default template is created during module setup, called `indexSuccess.php`.

And the `validate` directory is dedicated to configuration files used for form validation.

The directories `config`, `lib` and `validate` are empty for a new module.

Web tree structure

There are very few constraints for the `web` directory, but following a few basic naming conventions will provide default behaviors and useful shortcuts in the templates.

Conventionally, the static files are distributed in the following directories:

- `css/`: stylesheets with a .css extension
- `js/`: javascripts with a .js extension
- `images/`: images with a .jpg, .png or .gif format

The files uploaded by the users have to be stored in the `uploads` directory. Even though, most of the time, the `uploads` directory contains images, it is distinct from the image directory so that the synchronization of the development and production environments does not affect the uploaded images.

Tree structure customization

Even if it is highly recommended to keep the default tree structure, it is possible to modify it for specific needs, for instance in order to match the requirements of a client who already has its own tree structure and coding conventions.

Every path to a key directory is determined by a parameter ending with `_dir`. These parameters are defined in the framework, in a file called `$pear_data_dir/symfony/config/constants.php`, used by every symfony project. To customize the tree structure of an application, you simply need to override these settings. For that, use the `config.php` file located in the `apps/myapp/config/` directory (for an application-wide directory structure), or the `config/config.php` (for a project-wide structure).

You will probably have to copy the default configuration from the symfony directory if you need to modify it. Here is an extract of the default configuration:

```
// root directory structure
'sf_cache_dir_name'   => 'cache',
'sf_log_dir_name'     => 'log',
'sf_lib_dir_name'     => 'lib',
'sf_model_dir_name'   => 'model',
'sf_web_dir_name'     => 'web',
'sf_data_dir_name'    => 'data',
'sf_config_dir_name'  => 'config',
'sf_apps_dir_name'    => 'apps',

// global directory structure
'sf_app_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.$sf_app,
'sf_model_dir'      => $sf_root_dir.DIRECTORY_SEPARATOR.'model',
'sf_lib_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'lib',
'sf_web_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'web',
'sf_upload_dir'     => $sf_root_dir.DIRECTORY_SEPARATOR.'web'.DIRECTORY_SEPARATOR.'uploads',
'sf_base_cache_dir' => $sf_root_dir.DIRECTORY_SEPARATOR.'cache'.DIRECTORY_SEPARATOR.$sf_app,
'sf_cache_dir'      => $sf_root_dir.DIRECTORY_SEPARATOR.'cache'.DIRECTORY_SEPARATOR.$sf_app.DIREC
'sf_log_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'log',
'sf_data_dir'       => $sf_root_dir.DIRECTORY_SEPARATOR.'data',
'sf_config_dir'     => $sf_root_dir.DIRECTORY_SEPARATOR.'config',
```

Names and paths are both defined here. While this file is not that easy to read, it is easy to modify.

> **Important**: It is strongly recommended to use these parameters (via the `sfConfig` object) instead of their values when building and coding a project. That way, the applications created can be independent of the tree structure, more particularly for the first level directories.

```
[php]
 // always prefer
 if(file_exists(sfConfig::get('sf_log_dir').'myLogFile.log'))
 // instead of
 if(file_exists('/home/myproject/log/myLogFile.log'))
```

For instance, if you want your file structure to look like this (it is a standard shared host structure):

```
cgi-bin/
  apps/
    fo/
    bo/
  batch/
  cache/
  config/
  data/
    sql/
  doc/
    api/
  lib/
    model/
  log/
  test/
public_html/
  css/
  images/
  js/
  uploads/
  index.php
```

Change the `SF_ROOT_DIR` constant defined in the `index.php` (and the other front controllers) from:

```
define('SF_ROOT_DIR', realpath(dirname(__FILE__).'/..'));
```

to

```
define('SF_ROOT_DIR', realpath(dirname(__FILE__).'/../../cgi-bin'));
```

Then add the following lines to your `config/config.php`:

```
$sf_root_dir = sfConfig::get('sf_root_dir');
sfConfig::add(array(
  'sf_web_dir_name' => $sf_web_dir_name    = 'public_html',
  'sf_web_dir'      => $sf_root_dir.DIRECTORY_SEPARATOR.'..'.DIRECTORY_SEPARATOR.$sf_web_dir_name
  'sf_upload_dir'   => $sf_root_dir.DIRECTORY_SEPARATOR.'..'.DIRECTORY_SEPARATOR.$sf_web_dir_name
));
```

Configuration explained

Overview

To be simple and easy to use, symfony employs a default configuration that should answer the most common needs of standard applications without modification. However, it is possible to customize almost everything about how the framework and your application interact with the help of a set of simple and powerful configuration files. With these files, you will also be able to add specific parameters for your applications.

Introduction

The great advantage of web application frameworks is that they give developers full control over many aspects of a web application using configuration files - not hard-to-read code. No need to look for a specific class to change the logging behavior or to transform the way the URLs appear: just write some configuration, and you're done.

However, this approach has two serious drawbacks:

- developers end up writing endless hard-to-type XML files
- in a PHP architecture, every request takes much longer to process

Symfony uses configuration files only for what they are best at doing. As a matter of fact, the ambition of the configuration system in symfony is to be:

- **powerful**: almost every aspect that can be managed using configuration files is managed using configuration files
- **simple**: many aspects of configuration are not shown in a normal application, since they rarely need to be changed
- **easy**: configuration files are easy to read, to modify and to create by the final user (the application developer)
- **customizable**: the default configuration language is YAML, but it can be INI or XML or whatever the preference of the developer is
- **fast**: the configuration files are never processed by the application but by the configuration system, which compiles them into a fast-processing chunk of code for the PHP server

This is not just wishful thinking, since the PHP implementation of the configuration system actually allows it.

Why YAML ?

Symfony uses by default the YAML format instead of more traditional INI or XML formats. This decision is based on the great advantages of YAML. Below are some of the pros taken from the YAML website:

- YAML documents are very readable by humans.
- YAML interacts well with scripting languages.
- YAML uses host languages' native data structures.
- YAML has a consistent information model.

- YAML enables stream-based processing.
- YAML is expressive and extensible.
- YAML is easy to implement.

Contrary to INI, it supports hierarchy. Contrary to XML, it doesn't use a complicated syntax with lots of <, > and closing tags. The only disadvantage of YAML is that it is not widely known by developers, for now, as compared to the other formats. However, it is so simple to learn that it would be a pity to miss out on its features just for a lack of fame. As of now, if you are not familiar with the YAML syntax, you should get started right away.

For those who are attached to other configuration file formats, be reassured: symfony supports any type of configuration file, since they are processed by special handler that can be easily adapted (see the *Configuration handlers* section at the end of this chapter).

YAML syntax and symfony conventions

There are a few conventions that need to be kept in mind when writing YAML files:

- Never use tabs, use spaces instead. The YAML parsers won't be able to understand files with tabs, so indent your lines with spaces (a double blank is the symfony convention for indentation)

```
# Never use tabs
all:
-> mail:
-> -> webmaster:  webmaster@mysite.com


# Use blanks instead
all:
  mail:
    webmaster: webmaster@mysite.com
```

- To define an array as a value, enclose the elements in square brackets or use the expanded syntax with dashes:

```
# shorthand syntax for arrays
all:
  params: [value1, value2]


# expanded syntax for arrays
all:
  params:
    - value1
    - value2
```

- If your parameters are strings starting or ending with spaces, enclose the value in single quotes. If a string parameter contains special characters, also enclose the value in single quotes:

```
# strings starting or finishing with spaces are to be enclosed in single quotes
all:
  param1: This is a normal string
  param2: '  This is a special string '
  param3: 'That''s all folks!'
```

- To give a boolean value, use either `on`, `1` or `true` for a positive value and `off`, `0` or `false` for a negative one.

```
# Boolean values
true_values:
  - on
  - 1
  - true
false_values:
  - off
  - 0
  - false
```

The YAML website has an extensive description of the YAML syntax.

Environments

As a reminder, symfony provides three default environments : production (**prod**), test (**test**) and development (**dev**); you can add as many custom environments as you wish.

The different environments share the same php code (apart from the front web controller), but can have completely **different configuration** files. Consequently, each environment can connect to a different database.

In the development environment, the logging and debugging settings are all set to on, since maintenance is more important than performance. On the contrary, the production environment has settings optimized for performance by default, so the production configuration turns many features off.

The case of the test environment is a little special. This environment is used when you execute some code from your project on the command line, and that would mostly happen for unit testing. Consequently, the test environment is close to the production configuration, but it is not accessed through a web browser. It simulates the use of cookies and other HTTP specific components.

To add a new environment, you don't need to create a directory or to use the `symfony` command. Simply duplicate the production front controller (`myproject/web/index.php`) to a file called, say, `index_myenv.php`, and change the `SF_ENVIRONMENT` definition to `myenv` in this file. That's it, you have your new environment. As you will see, this environment inherits all the default configuration plus the settings that are common to all environments.

Configuration levels

There are several configuration levels in symfony:

- granularity levels
  - The **default configuration** located in the framework
  - The global configuration for the whole **project** (in `myproject/config/`)
  - The local configuration for an **application** of the project (in `myproject/apps/myapp/config/`)

        - ♦ The local configuration restricted to a **module** (in `myproject/apps/myapp/modules/mymodule/config`)
    - environment levels
        - ♦ specific to one environment
        - ♦ for all environments

Of all the properties that can be customized, many are environment-dependent. So many YAML configuration files are divided by environment, plus a tail section for **all** environments. Typical symfony configuration files result looking like:

```
# production environment settings
 prod:
   ...

 # development environment settings
 dev:
   ...

 # test environment settings
 test:
   ...

 # custom environment settings
 myenv:
   ...

 # setting for all environments
 all:
   ...
```

In addition, the framework itself defines default values in files that are not located in the project tree structure, but in the `$data_dir/symfony/config/` directory of your symfony installation. These settings are inherited by all applications:

```
# default settings:
 default:
   ...
```

Part of these default definitions are repeated in the project/app/module configuration files as comments (prefixed with `#`), so that you know that some parameters are defined by default and that they can be modified:

```
all:
 #  default_module:         default
 #  default_action:         index
 #
 #  error_404_module:       default
 #  error_404_action:       error404
 #
 #  login_module:           default
 #  login_action:           login
```

This means that a property can be defined several times, and the actual value results from a definition cascade with the following priority:

1. module
2. application
3. project
4. specific environment
5. all environments
6. default

Not all configuration files are environment-dependent.

Structure

To improve readability of the configuration files, symfony puts the parameter definitions in categories. Here is an example `app.yml`:

```
all:
   .general:
     tax:               19.6

   pager:
     max_per_page:      5

   mail:
     webmaster:         webmaster@mysite.com
     commercial:        commerce@mysite.com
```

The `.general`, `pager` and `mail` lines are category headers. Headers starting with `.` are only for readability and information, whereas the others define a **name space**. A name space allows you to use the same key name for several parameters, provided that they don't share the same name space. This is especially useful when you need to access the values with constants.

Access configuration from code

All the configuration files are eventually transformed into PHP, but some of the parameters need to be accessible from your code. In this case, they are available through a special `sfConfig` class. It is a registry for configuration parameters, with simple getter and setter class methods, accessible from every part of the code:

```
sfConfig::set('param_name', $value);         // to define a config
parameter = sfConfig::get('param_name', $default_value); // to retrieve a config parameter
```

So symfony configuration parameters have all the advantages of PHP constants, but without the disadvantages, since the value can be changed.

For instance, if you need to access the values defined in the previous example from your PHP code, write:

```
sfConfig::get('app_tax');
sfConfig::get('app_pager_max_per_page');
sfConfig::get('app_mail_webmaster');
sfConfig::get('app_mail_commercial');
```

The name is the concatenation of:

- a prefix related to the configuration file name (`sf_` for `settings.yml`, `app_` for `app.yml`, `mod_` for `module.yml`)
- the name space (if it is defined), in lower case
- the name of the key in lower case

The environment is not included, since your PHP code will only have access to the values defined for the environment it's executed in.

How to add a custom setting?

There are two ways of adding custom settings to your code:

- Add new lines to the usual configuration files
  - `myproject/apps/myapp/config/app.yml` for application wide settings
  - `myproject/apps/myapp/modules/mymodule/module.yml` for module wide settings (this files doesn't exist by default)
- Create a new configuration file, together with a new config handler

For instance, if you need to add the credit card operators accepted in you site, you can write in the application `app.yml`:

```
all:
   creditcards:
     fake:             off
     visa:             on
     americanexpress:  on

 dev:
   creditcards:
     fake:             on
```

To know if the `fake` credit cards are accepted in the current environment, get the value of:

```
sfConfig::get('app_creditcards_fake');
```

But maybe you need to add an array, an associative array or a lot of new parameters. In this case it is better to create a new configuration file, together with the corresponding configuration handler.

> **Note**: For project-wide custom settings, you can still take advantage of the `config/config.php` file to call directly the `sfConfig::set()` method or load your own YAML file with `sfYaml::load($file)`.

Scriptable configuration

It may happen that your configuration relies on external parameters (such as a database, or another configuration file). For these particular cases, the symfony configuration files are parsed as a PHP file before being passed to the YAML parser. It means that you can write:

```
all:
  translation:
    format:  <?php echo sfConfig::get('sf_i18n') == true ? 'xliff' : 'none' ?>
```

That's right: *you can put PHP code in YAML files*!

Beware though that the configuration is parsed very early in the life of a request, so you will have no symfony built-in methods or functions to help you. If you want one configuration file to rely on another, you have to make sure that the file you rely on is parsed before (look in the symfony source to find in which order the configuration files are parsed). The `app.yml` is one of the last files parsed, so you may rely on others in it.

Be also aware that in the production environment, the configuration is cached, so the configuration files are parsed (and executed) only once after the cache is cleared.

And if you are allergic to YAML, or if you prefer to avoid writing both a configuration file and a configuration handler to finally get an associative array in PHP, you can bypass YAML parsing completely and return a PHP array in any YAML file. It means that you can write the `logging.yml` as follows:

```
<?php
return array(
  'default' => array(
    'active' => true,
    'level'  => 'debug'
  )
);
```

...instead of the classic

```
default:
  active: on
  level:  debug
```

Configuration handlers

The configuration files, whatever their format, are processed by some special classes called **handlers** that transform them into fast-processing PHP code. To promote interactivity in the development environment, the handlers parse the configuration at each request so that you can see immediately a change in a YAML file. But of course, in the production environment, the processing occurs once during the first request, and then the processed PHP code is stored in the cache. The performance is guaranteed since every request in production will just execute some well-optimized PHP code.

For instance, if your `app.yml` contains:

```
all:
   mail:
     webmaster:         webmaster@mysite.com
```

Then the file `config_app.yml.php`, located in the `cache` folder of your project, will contain:

```
<?php
sfConfig::add(array(
  'app_mail_webmaster' => 'webmaster@mysite.com',
```

```
));
?>
```

This is a major advantage over many PHP frameworks, where configuration files are compiled at every request even in production. Unlike Java, PHP doesn't share an execution context between requests. For other PHP frameworks, keeping the flexibility of XML configuration files requires a major performance hit to process all the configuration at every request. This is not the case in symfony. Thanks to the cache system, the overhead caused by configuration is very low.

The other main advantage of using configuration handlers is that it allows you to use formats other than YAML. The configuration handlers just transform configuration files into optimized PHP code. If you have a parser for another language - and PHP has native support of XML and INI files - it is very easy to write a custom handler that will read your own configuration file format.

Wrap up

To sum up, just remember the syntax of YAML configuration files:

```
environment:
   .header1:
     key1:      value1
     key2:      value2

   namespace1:
     key3:      value3
     key4:      value4

   # comment
```

But now, enough of these general considerations, let's dig into the real configuration files and all their marvelous possibilities.

Configuration in practice

Overview

The configuration can be defined at several levels : project, application, module. It can also be defined for several environments : development, test, production, and any additional environment needed. By modifying YAML files, you get the power of configuration at your fingertips.

Introduction

The symfony configuration system is inspired by the way Mojavi handles configuration. Although not appreciated by everyone, this system is extremely powerful.

Configuration is distributed into files, by subject. The files contain parameter definitions, or settings.

The parameters can be set in a cascade of definitions. For instance, a given parameter can be defined at the project level, redefined at the application level, and further redefined at the module level. But this cascading also applies to environments. A parameter definition in a named environment has precedence over the same parameter definition for `all` environments, which has precedence over a definition in the default configuration.

> **Note**: Environment-dependent configuration files will be identified by a star `*` in this documentation.

Project configuration

There are a few project configuration files by default. Here are the files that can be found in the `myproject/config/` directory:

- `apache.conf`: This file is not really a configuration file, since it is not used by the project. It is given as an example of an Apache configuration for a typical symfony project.
- `config.php`: Can hold general project configuration. If you add some `define` statements in this file, the constants will be accessible from every application of the project. This file is empty by default. Can also be used to override the default project file structure. This file is empty by default. *see more below*
- `properties.ini`: Holds a few parameters used by the pakefile (including the project name)

- `rsync_exclude.txt`: this file specifies which directories have to be excluded from the synchronization between environments. The default content should illustrate its use very well:

```
stats
.svn
web/uploads
cache
log
web/index.php
config
```

- `schema.yml` and `propel.ini` are data access configuration files used by Propel (symfony's ORM layer). They are used to plug the Propel libraries with the symfony classes and the data of your project. The `schema.yml` contains a representation of the project's relational data model. The `propel.ini` is automatically generated, so you probably do not need to modify it. If you don't use Propel, these files are not needed.

Application configuration

The principal part of the configuration is the application configuration. The files located in `myproject/apps/myapp/config/` will be briefly described here before a deeper look at how to modify them.

Overview

- `app.yml*`: This file should contain the application-specific configuration, i.e. global variables that don't really need to be stored in a database. VAT rates, shipping fares, email addresses are often stored in this file. It is empty by default. See more in the configuration chapter.
- `config.php`: This file bootstraps the application, which means that it does all the very basic initializations to allow the application to start. It normally defines the include path to the framework libraries, defines the directory layout (by including the `constants.php` file), includes the project configuration (`myproject/config/config.php`), loads the necessary symfony classes, and includes some of the parsed `.yml` files of the application configuration (the others are loaded on demand). This configuration file is written in PHP rather than YAML because the YAML interpreter isn't loaded when `config.php` is processed.
- `databases.yml*`: This is where you define the access and the connection settings to the database (host, login, password, database name). *see more below*
- `factories.yml*`: By default, the symfony framework uses some specific classes for its operation; this file allows you to override this behavior by pointing to alternate classes to manage sessions, actions, front web controller, etc... You normally wouldn't change the content of this file, as will be explained a little further. *see more below*
- `filters.yml`: Symfony allows the execution of filters before actions. For instance, the Security filter is configured by default to check credentials for restricted actions. If you need to add a custom filter, for instance to calculate the time to execute an action, this is the file that you need to modify.
- `logging.yml*`: Defines which level of detail has to be recorded in the logs, to help you supervise and debug your application. See below for more details. *see more below*
- `routing.yml`: The routing rules, that allow transforming unreadable and unbookmarkable URLs into "smart" and explicit ones, are stored in this file. For new applications, a few default rules exist. See more in the routing chapter.

- `settings.yml*`: The main settings of a symfony application are defined in this file. This is where you specify if your application has internationalization, its default language, the request timeout, whether caching is turned on or not, and whether routing is turned on. With a one line change in this file, you can shut down the application so you can perform maintenance or upgrade one of its components. This is a perfect example of the benefit of using a single front web controller. *see more below*
- `tidy.yml*`: If the use of HTML Tidy is activated in the `settings.yml`, then you can modify the options of this utility to alter the HTML code output to, for instance, re-indent the tags properly, or remove comments, or collapse all spaces and carriage returns to save bandwidth, or to correct missing closing tags that were missed by the developers.
- `view.yml`: The structure of the default View (name of the layout, title and metas, default .js and .css files to be included, name of the included slots, etc.) is set in this file. These settings can be overridden for each module. This file also defines the default value of the `meta` and `title` tags. *see more below*

`settings.yml`: General settings

The `settings.yml` file, which is environment dependent, contains the main application configuration. Here is the beginning of its default content:

```
prod:

dev:
  .settings:
    # E_ALL | E_STRICT = 4095
    error_reporting:        4095
    web_debug:              on
    cache:                  off
    stats:                  off
    no_script_name:         off

test:
  .settings:
    cache:                  off
    stats:                  off
    web_debug:              off

all:
#  .actions:
#    default_module:         default
#    default_action:         index
#
#    error_404_module:       default
#    error_404_action:       error404
#
#    login_module:           default
#    login_action:           login
#
#    module_disabled_module: default
#    module_disabled_action: disabled
#
#    secure_module:          default
#    secure_action:          secure
#
#    unavailable_module:     default
```

```
#    unavailable_action:     unavailable
#
#  .settings:
#    available:              on
#    module_accessor:        module
#    action_accessor:        action
#    content_type:           html
...
```

First of all, you may notice that most of the configuration is inherited from the `default` definition (the statements starting with a `#` in the configuration for `all` environments are **comments**) It means that, for these parameters, the default configuration is used instead. If you need to override any of them, just remove the comment `#` marker at the beginning of the appropriate line and change the value.

The parameters defined in the `.action` category identify the modules and actions to be used under certain circumstances. In particular:

- `default_*`: Specifies which action of which module has to be called when not specified in the URL. This is especially useful to set the home page action of your website (the one that will be called with the relative URL '/')
- `error_404_*`: Specifies the default module/action to be called when a 404 error (page not found) occurs. It defaults to `default/error404`. This action is not explicit in a new application, but you can override it or choose a completely different module/action
- `login_*`: When a secure page requiring credentials is accessed by an anonymous user, the user will be automatically redirected to a login page. This parameter defines the module/action to use for login purposes.

Here is a list of some of the useful `.settings` parameters:

- `available`: When set to `off`, it shuts down the whole application. The user will see a *Application temporarily unavailable for maintenance* type of message
- `use_database`: If your application doesn't use a database, this parameter should be set to `off`
- `use_security`: If your application has restricted areas, authentication and credentials, set this parameter to `on` (see the security chapter for more details)
- `compressed`: Activates the HTML compression to reduce bandwidth requirements and improve performance
- `tidy`: Activates the use of HTML Tidy
- `i18n`: Must be set to `on` for sites/applications available in multiple languages
- `default_culture`: Specifies the default parameter used to format dates, numbers, currencies (`en` in the default configuration)
- `web_debug`: Activation of the web debug frame, a tool that gives access to debug info on every page. The quantity of logged information is relative to the entries set in the `logging.yml` file, and it requires that the `SF_DEBUG` constant be set to `true` in the front controller. See more in the debug chapter.
- `cache`: Activates the caching feature to speed up page generation by recording chunks of compiled code
- `routing`: Activates the routing feature to transform the outputted URLs and allow "smart" URLs to be interpreted. See more in the routing chapter.
- `stats`: Activates the recording of statistics for the application

Remember that each one of these settings is accessible from inside the PHP code via the `sfConfig` class, as explained in the configuration chapter. The parameter name is the setting name prefixed with `sf_`. For instance, if you want the value of the `cache` parameter, you just need to call `sfConfig::get('sf_cache')`.

`databases.yml`: Database settings

If the application/site uses a database, you have to configure the database access by entering a set of parameters in the `databases.yml` file.

```
all:
  propel:
    class:          sfPropelDatabase
    param:
      datasource:   symfony
      dsn:          mysql://root:@localhost/mydatabase
```

The above example is a shorthand syntax for the setting of all the data access parameters, the long form is shown below:

```
all:
  propel:
    class:          sfPropelDatabase
    param:
      datasource:   symfony
      phptype:      mysql
      hostspec:     localhost
      database:     mydatabase
      username:     root
      password:
```

To learn more about data binding and the access to a database, go to the data access chapter.

> **Note**: If your application doesn't use a database, you can improve its performance by setting `use_database` to `off` in the `settings.yml`.

`logging.yml`: Logging settings

Symfony offers two ways to watch the log messages.

Classically, the logs are written in files. Symfony stores message logs in files according to the application and the environment. For instance, in the `myproject/log/` directory, you will probably find two files:

```
myapp_dev.log
myapp_prod.log
```

Don't forget to periodically rotate these files, since symfony will not do it automatically.

If the `web_debug` feature is set to `on` for your application (in `settings.yml`), the logs for each request are also available in the browser, in a special layer that appears on the right of the screen. Note that this option is activated by default in the development environment. Refer to the debug chapter for more information.

The `logging.yml` file defines the level of log messages recorded. By default, all levels are included (from alerts to unrecoverable errors). In the production environment, only the errors are logged.

```
prod:
  level:    err

dev:

test:

all:
#  active:  on
#  level:   debug
```

`factories.yml`: Factories settings

Symfony uses classes such as `sfFrontWebController`, `sfRequest`, `sfUser`, that are part of the framework. In the `myproject/apps/myapp/lib/` directory, you can override them with your own `myFrontWebController`, `myRequest` classes (`myUser` already exists). They simply need inherit of the 'sf-' classes, and this allows you to customize their behavior. Teel the framework to use the 'my-' classes instead of the 'sf-' ones by changing the settings found in the `factories.yml` file:

```
...
all:
#  controller:
#    class: sfFrontWebController
#
#  request:
#    class: sfWebRequest
#
#  user:
#    class: myUser
...
```

Have a look at the existing `myUser` class to see how the inheritance allows for extension and overriding of **factories** classes.

Front controller configuration

The very first application configuration is actually found in the **front controller**. Take a look at this default `web/index.php`:

```
<?php

define('SF_ROOT_DIR', dirname(__FILE__).'/..');
define('SF_APP', 'fo');
define('SF_ENVIRONMENT', 'prod');
define('SF_DEBUG', true);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATO

sfContext::getInstance()->getController()->dispatch();

?>
```

After defining the name of the application (`fo`) and the environment (`prod`), the general configuration file is called before the dispatching. So a few useful constants are defined here:

- `SF_ROOT_DIR`: Project root path (normally, should remain at its defaults value `dirname(__FILE__).'/..'`)
- `SF_APP`: Application name in the project
- `SF_ENVIRONMENT`: Environment name (`prod`, `dev` or any other project-specific environment that you define)
- `SF_DEBUG`: Activation of the debug mode

If you need to change one of these values, you probably need an additional front controller. The controller chapter will tell you more about it.

Additional application configuration

A second set of configuration files is in the symfony installation directory; the settings defined there are default settings that seldom need to be modified, or that are global to all projects. However, if you need to modify them, just copy the required file from the `$pear_data_dir/symfony/config/` directory to your `myproject/apps/myapp/config/` directory. The settings defined in an application always have precedence.

- `autoload.yml`: Settings of the autoloading feature. This feature exempts you from requiring custom classes in your code if they are located in specific directories. *see more below*
- `constants.php`: Overrides the default application file structure. *see more below*
- `core_compile.yml` and `bootstrap_compile.yml` are lists of classes to be included to start an application (in `bootstrap_compile.yml`) and to process a request (in `core_compile.yml`). These classes are actually concatenated into an optimized PHP file without comments, which will accelerate the execution by minimizing the file access operations (1 file is loaded instead of more than 40 for each request). This is specially useful if you don't use a PHP accelerator.
- `config_handlers.yml`: Do you remember about configuration handlers? This is where you can add or modify the handlers used to process each configuration file, for instance to use the less flexible INI or XML files instead of the more efficient YAML files. Each configuration file having a handler class definition, you will find the customization quite straight forward.
- `php.yml`: Checks that the variables of the `php.ini` file are properly defined, and allows you to override them if necessary. *see more below*

`autoload.yml`: Autoloading settings

When the PHP parser encounters a `new myClass` statement in the code of an application, it looks for a 'myClass.class.php' file in the paths defined in the `autoload.yml`. If the file is found, the framework loads the required library and includes it automatically.

In other words, the autoloading feature exempts you from requiring custom classes in your code, provided that they are located in a directory defined in the `autoload.yml`. This means that you don't need to require libraries at the top of your classes, just let the framework do the job for you, and it will load only the necessary classes and at the appropriate time.

By default, the autoloading works for classes that are located in the following directories:

- `myproject/lib/`,
- `myproject/lib/model`,
- `myproject/apps/myapp/lib/` and
- `myproject/apps/myapp/modules/mymodule/lib`.

Here is an extract of the default `autoload.yml` (found in `$pear_data_dir/symfony/config/autoload.yml`):

```
autoload:
  symfony_core:
    name:           symfony core classes
    ext:            .class.php
    path:           %SF_SYMFONY_LIB_DIR%/symfony
    recursive:      on

  symfony_orm:
    name:           symfony orm classes
    files:
      Propel:       %SF_SYMFONY_LIB_DIR%/symfony/addon/propel/sfPropelAutoload.php
      Criteria:     %SF_SYMFONY_LIB_DIR%/propel/util/Criteria.php
      SQLException: %SF_SYMFONY_LIB_DIR%/creole/SQLException.php
      DatabaseMap:  %SF_SYMFONY_LIB_DIR%/propel/map/DatabaseMap.php

  project:
    name:           project classes
    ext:            .class.php
    path:           %SF_LIB_DIR%
    recursive:      on
    exclude:        [model, plugins]

  ...
```

Each autoloading rule has a label, both for visual organization and ability to override it.

If you want to autoload classes stored somewhere else in your file structure, you need to create a new `autoload.yml` in your `myproject/apps/myapp/config/` folder and edit it. You can either override existing rules or add new ones. The custom extension chapter will tell you more about it.

`php.yml`: PHP configuration

In order to have a php environment compatible with the rules and best practices of agile development, symfony checks and overrides a few settings of the `php.ini` configuration. This is what the `php.yml` file is used for. Here is the default `php.yml` (found in `$pear_data_dir/symfony/config/`):

```
set:
  magic_quotes_runtime: off
  log_errors:           on
  arg_separator.output: \&amp;

check:
  magic_quotes_gpc:     off
  register_globals:     off
```

The variables defined under the `set` category are modified (despite how they were defined in the `php.ini`). The variables defined under the `check` category cannot be modified on the fly, so their values are checked and an exception raised if your current configuration doesn't match these criterias.

For instance, the default `php.yml` sets the `log_errors` to `on` so that you can trace errors in symfony projects. It also recommends the `register_globals` to be set to `off` to prevent security breaches.

If you don't want symfony to apply these settings, or if you want to run a project with `magic_quotes_gpc` and `register_globals` set to `on` (wich we strongly advise against), copy the default `php.yml` into your application `config` directory, and change the values to be set or checked. Alternatively, you can delete the lines that are not compatible with your environment in the application copy of the file.

Module configuration

By default, a module has no specific configuration. But, according to your requirements, you can override some application level settings for a given module, or add new parameters restricted to a specific module.

As you may have guessed, module configuration files have to be located in a `myproject/apps/myapp/modules/mymodule/config/` directory. These files are:

- `generator.yml`: To create a data access interface (useful for back-office generation) *see more below*
- `module.yml*`: Custom parameters specific to a module. *see more below*
- `security.yml`: Access restrictions for actions. *see more below*
- `view.yml`: Configuration for the views of one or all of the actions of a module. *see more below*
- data validation files: Although located in the `validate/` directory instead of the `config/` one, the YAML data validation files, used to control the data entered in forms, are also specific to a module. *see more below*

Most module configuration files offer the ability to define parameters for all the views or all the actions of a module, or for a subset of them.

`module.yml`: Custom module parameters

Modules can have their own settings. In such cases, these settings are defined in a `module.yml` file. For instance, a 'poll' module might need a `max_votes` parameter:

```
all:
   .settings:
     max_votes:      150
```

As mentioned in the configuration chapter, the parameter is accessible from the code with the following call:

```
$max_votes_parameter = sfConfig::get('mod_max_votes');
```

But maybe you need a totally specific configuration file. In such cases, as in the case of custom app configuration, create a `myconfig.yml` file together with a `config_handlers.yml` file to handle its data (read the configuration chapter for more information).

`security.yml`: Access restriction configuration

To restrict access to an action to a subset of authenticated users having specific credentials, you need to add a module configuration file called `security.yml`.

Here is an example module security configuration file:

```
update:
  is_secure:  on

save:
  is_secure:  on

orders:
  is_secure:  on

all:
  is_secure:  off
  credential: customer
```

The `update`, `save` and `orders` actions of this module will only work for authenticated users with the `customer` credential.

To learn more about security and the way to set credentials, review the security chapter.

`generator.yml`: generated module configuration

One commonly used module configuration file is the `generator.yml` file. When you setup a 'Create Read Update Delete' (CRUD) basic layout for a data object, the module created will have a `generator.yml` file that you can modify. Read more about CRUDs, scaffolding and generated administrations in the scaffolding chapter.

`validate.yml`: Form validation

The last common module configuration file relates to form validation. It is not located in the `myproject/apps/myapp/modules/mymodule/config/` directory but in the `myproject/apps/myapp/modules/mymodule/validate/` directory. YAML validation files look like the following:

```
methods:
  get:  [new_email, new_password, new_password2]
  post: [new_email, new_password, new_password2]

names:
  new_email:
    required:     Yes
    required_msg: Please enter an email address
    validators:   emailValidator

  new_password:
    required:     Yes
    required_msg: Please enter a password
    validators:   passwordValidator
```

```
  new_password2:
    required:     Yes
    required_msg: Please confirm your password

passwordValidator:
    class: sfStringValidator
    param:
      min:        4
      min_error:  Your password needs at least 4 characters
      max:        12
      max_error:  Your password can not have more than 12 characters

emailValidator:
    class: sfEmailValidator
    param:
      email_error:  The email you entered is not valid
```

For more information about form validation and the use of the validation configuration files, review the form validation chapter.

Multiple level configuration

`view.yml`: View configuration

The rules and parameters governing the View (interface components, layout, headers, etc.) are set in the `view.yml` configuration file. The file exists by default in the application configuration directory (`myproject/apps/myapp/config/`), and these settings can be overridden at the module level by adding a `view.yml` file to a `myproject/apps/myapp/modules/mymodule/config/` directory.

Refer to the chapter describing the view configuration to learn more about this feature.

Here is an example module view configuration file:

```
indexSuccess:
  javascripts:  [myinteraction]

indexError:
  title:        Sorry, but there is an error
  layout:       error
  stylesheets:  [error]

listSuccess:
  template:     listtemplate
  components:
    breadcrumb: []

all:
  layout:       mylayout
  components:
    navigation: [bar, navigation]
    breadcrumb: [bar, breadcrumb]
```

The default layout for all the actions of this example module is set to `mylayout.php` (as opposed to the default layout for the whole application, called `layout.php` by convention). The `navigation` and `breadcrumb` components slots are defined for all the actions of the module, except for the action `list` where the `breadcrumb` component slot is suppressed. For this action, the template to be used will not be `listSuccess.php` but `listtemplate.php`. If the result of the `index` action is `sfView::ERROR`, the template `indexError.php` will be integrated into the `error.php` layout, including a special `error.css` stylesheet and a custom page title. In addition, the `index` action requires a special javascript called `myinteraction.js` to be included in the page header so that the template can work correctly.

File structure settings

The `config.php` file is often used to customize the directory layout, the directory separator, and everything that the framework needs to find your actions (normally in `actions`), your modules (normally in `modules`), your libraries (normally in `lib`), etc.

Every path to a key directory is determined by a parameter ending with `_dir`. Here is an extract of the standard file structure configuration (located in `$pear_data_dir/symfony/config/constants.php`):

```
...
sfConfig::add(array(
  // root directory structure
  'sf_cache_dir_name'   => 'cache',
  'sf_log_dir_name'     => 'log',
  'sf_lib_dir_name'     => 'lib',
  'sf_model_dir_name'   => 'model',
  'sf_web_dir_name'     => 'web',
  'sf_data_dir_name'    => 'data',
  'sf_config_dir_name'  => 'config',
  'sf_apps_dir_name'    => 'apps',

  // global directory structure
  'sf_app_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.$sf_app,
  'sf_model_dir'      => $sf_root_dir.DIRECTORY_SEPARATOR.'model',
  'sf_lib_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'lib',
  'sf_web_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'web',
  'sf_upload_dir'     => $sf_root_dir.DIRECTORY_SEPARATOR.'web'.DIRECTORY_SEPARATOR.'uploads',
  'sf_base_cache_dir' => $sf_root_dir.DIRECTORY_SEPARATOR.'cache'.DIRECTORY_SEPARATOR.$sf_app,
  'sf_cache_dir'      => $sf_root_dir.DIRECTORY_SEPARATOR.'cache'.DIRECTORY_SEPARATOR.$sf_app.DIR
  'sf_log_dir'        => $sf_root_dir.DIRECTORY_SEPARATOR.'log',
  'sf_data_dir'       => $sf_root_dir.DIRECTORY_SEPARATOR.'data',
  'sf_config_dir'     => $sf_root_dir.DIRECTORY_SEPARATOR.'config',
  ...
```

To learn more about the default tree structure, refer to the file structure chapter.

You will probably need to modify this file if you develop an application for a client who already has a defined directory structure and who is not willing to change it to comply with the symfony logic (how could that be possible ?).

Projects inherit their file structure from the framework `constants.php`. This means that if you need to change the file structure of your project or application, you have to override the settings of the `constants.php`. This can be done in your project `config.php` file, to change the tree structure of all the

applications of the project, or in the application `config.php`, to change the file structure of a single application.

Beware that the application `config.php` is not empty, so if you need to include file structure definitions there, do it just after the inclusion of the project configuration.

Browsing your own YAML file

Whether you want to create a new config handler, or to read a YAML file directly, you can use the `sfYaml` class. It is a YAML parser that can turn a YAML file into an associative array:

```
$myarray = sfYaml::load('/tmp/myfile.yml');
```

MVC implementation explained

Overview

The symfony framework relies on the best practices of web development, and especially on the MVC model (Model/View/Controller). This chapter describes the general functioning of the symfony framework and the paradigm it relies on.

Concepts

The base of symfony is a classic web design pattern, the MVC architecture, which has three levels:

- the **Model** represents the information on which the application operates, its business logic
- the **View** renders the model into a web page suitable for interaction with the user
- the **Controller** responds to user actions, and invokes changes on the model or view as appropriate

In symfony, the Controller is separated in different parts, namely a front controller and a set of actions. Each action knows how to handle a specific kind of request. The View is separated in layouts, templates and partials - all those are PHP files that have access to the content defined in the Action. The Model offers an abstraction layer to databases and useful information about the session and the requests.

This model helps to work separately on the business logic (Model) and on the presentation (View). For instance, if you need an application to run both on standard web browsers and on handheld devices, you just need a new View but you can keep the original Controller and Model. The Controller helps to hide the detail of the protocol used for the request (HTTP, console mode, mail, etc.) from the Model and the View. And the Model abstracts the logic of the data, which makes the View and the Action independent of, for instance, the type of database used by the application.

Symfony implements the MVC model in a simple and light way so that developers get the benefits of shortcut conventions and agile programming without unnecessary slowdowns of the application itself.

Example

To make things clear, let's see how the symfony implementation of the MVC architecture works for an *add to cart* interaction.

1. First, the user interacts with the user interface by selecting a product and pressing a button; this probably validates a form and sends a web request to the server.
2. The Front Controller then receives the notification of the user action and, after executing general procedures on this request (regarding routing, security, etc.), understands that it need to be passed to the 'shopping cart' action.
3. The 'shopping cart' Action accesses the Model, to update the 'cart' object of the user session.
4. As the modification of the stored data is successful, the action prepares the content that will be included in the response - confirmation of the addition and complete list of products currently in the shopping cart. The 'shopping cart' action logic specifies that for a product addition, the content has to be included into a 'shopping cart' template.

5. The View then assembles the answer from the action and the skinning from the template to produce the HTML code of the shopping cart page.
6. It is finally transferred to the web server that sends it to the user, who will use his browser to read and interact with the new information.

Symfony classes

The names of the classes that are part of the symfony package are prefixed by `sf`. You can find these default classes in your symfony/lib installation directory.

```
sfAction
sfRequest
sfView
...
```

If you need to add custom methods to these classes for a given project, modify the `my`-prefixed classes that can be found in the `myproject/apps/myapp/lib` directory:

```
myAction
myRequest
myView
...
```

These classes inherit the `sf`-ones and support extension.

> **Note**: the names of the framework classes that will be called during execution are specified in the `factories.yml` application configuration file.

Symfony model

Overview

Symfony has an object/relational abstraction layer based on the Propel project. Accessing data stored in a database and modifying it is made easy through an object translation of the database. This chapter explains the creation of such an object data model, the way to access and modify the data in Propel. It also illustrates the integration of Propel in Symfony.

Why use an abstraction layer?

Databases are relational. PHP5 and symfony are object oriented. In order to access the database in an object-oriented way, an interface translating the object logic to the relational logic is required. It is called an object-relational mapping, or ORM.

This is the **Model** layer of the symfony MVC architecture. It is made up of objects that give access to data and keep business rules within themselves.

One benefit of an object/relational abstraction layer is that it prevents you from using a syntax that is specific to a given database. It automatically translates calls to the model objects to SQL queries optimized for the current database.

This means that switching to another database system in the middle of a project is easy. Imagine that you have to write a quick prototype for an application, but the client hasn't decided yet which database system would best suit his needs. You can start building your application with SQLite, for instance, and switch to MySQL, PostgreSQL or Oracle when the client is ready to. Just change one line in a configuration file, and it works.

An abstraction layer encapsulates the data logic. The rest of the application doesn't need to kow about the SQL queries, and the SQL that accesses the database is easy to find. Developers who are specialized in database programming also know clearly where to go.

Using objects instead of records, and classes instead of tables, have another benefit: They allow you to add new accessors to your tables. For instance, if you have a table called 'Client' with two fields 'FirstName' and 'LastName', you might like to be able to require just a 'Name'. In an object-oriented world, it is as easy as adding a new accessor method to the `Client` class.

```
public function getName()
{
  return $this->getFirstName.' '.$this->getLastName();
}
```

All the repeated data access functions and the business logic of the data itself can be kept in such objects. Imagine a class 'ShoppingCart' in which you keep items (which are objects). To get the full amount of the shopping cart for the checkout, you can add the following method:

```
public function getTotal()
{
  $total = 0;
```

```
  foreach ($this->getItems() as $item)
  {
    $total += $item->getPrice();
  }
  return $total;
}
```

And that's it. Imagine how long it would have required to write a SQL query doing the same?

Propel is currently one of the best object/relational abstraction layer for PHP5. Instead of rewriting it, symfony integrates it seamlessly into the framework.

Data model

Purpose

In order to create the data object model that symfony will use, you need to translate whatever relational model your database has to an object data model. This is done through a schema file called `schema.yml`, in which are defined the tables, their relations and the type of their columns.

The `schema.yml` file must be located in the `myproject/config/` directory. This file uses the YAML syntax, but it is simple to read without any knoledge of that format (which, by the way, takes 5 minutes to learn).

Example

Now, let's imagine that you have a database 'blog' already containing data in two tables: 'blog_article' and 'blog_comment' with the following structure:

blog_article	**blog_comment**
id	id
title	article_id
content	author
created_at	content
	created_at

The related `schema.yml` should simply look like:

```
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:
```

Notice that the name of the database itself (`blog`) doesn't appear in the `schema.yml`. Instead, the database is described under a connection name (`propel` in this example). This is because the actual connection settings can depend on the environment your application runs in. For instance, when you run your application in development, you will access a development database, but with the same schema as the production database, which has different connection settings. These settings will be specified in the `databases.yml` (see below).

Basic schema syntax

In a `schema.yml`, the first key represents a connection. It can contain several tables, each having a set of columns. According to the YAML syntax, the keys end with a colon, and the structure is shown through indentation (one or more spaces, but no tabulations).

A table can have special attributes, including the `phpName` describing the name of the class that will be generated. If you don't mention it, the name of the table will be used in place, after removing underscores and capitalizing the first letter of inner words (in the example, the default `phpName` of the tables would be `blogArticle` and `blogComment`). Columns also have a `phpName`, which is the capitalized version of the `name` (`ID`, `TITLE`, `CONTENT`, etc.) and doesn't need overriding in most cases.

Most of the time, columns only need one attribute: the type. It can take the usual values: `boolean`, `integer`, `float`, `date`, `varchar(size)`, `longvarchar`, etc. For text content over 256 characters, you need to use the `longvarchar` type, which has no size but can not exceed 65Kb (MySQL specific). Note that the `date` and `timestamp` types have the usual limitations of Unix dates and can not be set to a date prior to 1970-01-01. As you may need to set older dates (for instance for dates of birth), a format of dates 'before Unix' can be used with `bu_date` and `bu_timestamp`.

Of course, if you want to specify your own primary keys, foreign-keys, default values or index, you can use the extended schema syntax (see below)

Schema conventions

Columns called `id` are considered to be primary keys.

Column ending with `_id` are considered to be foreign keys, and the related table is automatically determined according to the first part of the column name.

Columns called `created_at` are automatically set to the `timestamp` type.

For all these columns, you don't need to specify any type, since symfony will deduce their format from their name. That is why the `schema.yml` is so easy to write.

Object data model files

Code generation

Now that the data schema is described, the model classes are ready to be generated. In your project directory, type the command:

```
$ symfony propel-build-model
```

> **Note**: Propel uses Phing at this point, so you will need to install it if it wasn't done already by calling:
>
> ```
> $ pear install http://phing.info/pear/phing-current.tgz
> ```

The base data access classes will be automatically created in the `myproject/lib/model/om/` directory:

```
BaseArticle.php
BaseArticlePeer.php
BaseComment.php
BaseCommentPeer.php
```

In addition, the actual data access classes will be created in `myproject/lib/model`:

```
Article.php
ArticlePeer.php
Comment.php
CommentPeer.php
```

You only defined two tables and you end up with eight files. What's the deal here?

Base and current model

Why keep two versions of the data object model, one in `model/om/` and another in `model/`?

You will probably need to add custom methods and attributes to the model objects (think about the `->getName()` method that outputs the `FirstName` and `LastName` together). But as your project develops, you will also add tables or columns. Whenever you change the `schema.yml`, you will have to regenerate the object model classes by making a new call to `symfony propel-build-model`. If your custom methods were written in the classes actually genereated, then it would be erased after each generation.

The `Base` classes kept in the `model/om/` directory are the ones generated by Propel. You should **never** modify them since every new build of the model will completely erase these files. But the regular object classes, kept in the `model/` directory, actually inherit from the previous ones, and are never overwritten. So this is where you can add custom methods.

For example, here is the content of the newly created `model/Article.php` file:

```
require_once 'model/om/BaseArticle.php';
class Article extends BaseArticle
{
}
```

It inherits all the methods of the `BaseArticle` class, but a modification in the model will not affect it. This structure is both customizable and evolutionary.

This mechanism allows you to start coding even without knowing the final relational model of your database.

Peer classes

The classes `Article` and `Comment` will allow you to access attributes of a record of the related tables. This means that you will be able to know the title of an article by calling a method of an `Article` object:

```
$article = new Article();
...
$title = $article->getTitle();
```

But there is more to databases than reading what's inside records: You also need to actually find records. That's the purpose of the `Peer` classes. They offer class methods to find records, that return an object or a lists of objects of the related 'no `Peer`' class.

```
$articles = ArticlePeer::retrieveByPks(array(123, 124, 125));
// $articles is an array of objects of class Article
```

From a data model point of view, note that there cannot be any `Peer` object. That's why the methods of the `Peer` classes will be called with a `::` (for class method call) instead of the usual `->` (for object method call).

Data access

In Propel, your data is accessed through objects. If you are used to the relational model, and to SQL to retrieve and alter your data, the Propel methods will probably look complicated. But once you've tasted the power of object orientation for data access (have a look at the examples given in the Propel documentation), you will probably like it a lot.

Field access

Propel provides one class for each the tables defined in the `schema.yml` file. These classes have default creators, accessors and mutators: The `new`, `get` and `set` methods give access to the columns of the table.

```
$article = new Article();
$article->setTitle('My first article');
$article->setContent('This is my very first article.\n Hope you enjoy it!');

$title   = $article->getTitle();
$content = $article->getContent();
```

Note that the `phpName` attribute of the `table` is used for the class name; the accessors use a CamelCase variant of the column names in which the first letter of each word is capitalized and underscores are suppressed.

To create a record in a table related to another, simply pass the object as a foreign key:

```
$comment = new Comment();
```

```
$comment->setAuthor('Steve');
$comment->setContent('Gee, dude, you rock: best article ever !');
$comment->setArticle($article);
```

The last line automatically sets the `ArticleId` attribute of the `Comment` object to the value of the `Id` attribute of the `$article` object. Of course, you could have done it manually:

```
$comment->setArticleId($article->getId());
```

Note that by calling the `new` constructor, you created a new object but not an actual record in the `Article` table. In order to save the data into the database, you need to `save()` your object:

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

The cascading principle of Propel exempts you from also saving the `$comment` object: As the record is linked to the one you saved, it will be saved automatically.

> **Hint**: To set several fields at one time, you can use the `->fromArray()` methods of the Propel objects:
>
> ```
> $article->fromArray(array(
>    'title'   =>   'My first article',
>    'content' =>   'This is my very first article.\n Hope you enjoy it!',
> ));
> ```

Access to related records

Let's check all the comments about your new article:

```
$comments = $article->getComments();
```

Propel sees the `article_id` column in the `Comment` table, and understands that an article can have many comments (one-to-many relationship). During the code generation, a `getComments()` method was created in the `Article` object to get the array of related Comments. Note the plural used here in the `get` method. So the `$comments` variable contains an array of objects of class `Comment`, and you can display the first one with:

```
print_r($comments[0]);
```

If you read comments to your articles, you might change your mind about the interest of publishing on the Internet. And if you don't appreciate the irony of article reviewers, you can easily delete the comments with the `delete()` method:

```
foreach($comments as $comment)
{
  $comment->delete();
}
```

For many-to-one relationships, it's even simpler:

```
echo $comment->getArticle()->getTitle();
```

The `getArticle()` method returns an object of class `Article`, which benefits from the `getTitle()` accessor. This is much better than doing the join yourself, which may take a few lines of code (starting from the `$comment->getArticleId()` call).

Find more about these methods and their syntax in the Propel documentation.

Records access

If you know the primary key of a particular record, use the `retrieveByPk()` class method of the `Peer` class of the table to get the related object. For instance:

```
$article = ArticlePeer::retrieveByPk(7);
```

The `schema.yml` defines the `id` field as the primary key of the `Article` table, so this statement will actually return the article that has `id` number 7. As you used the primary key, you know that only one record will be returned. So the `$article` variable contains an object of the `Article` class.

For simple queries, Propel doesn't use SQL, to allow database abstraction. Instead, it offers a `Criteria` object, which has the same power and is very easy to use.

For instance, to retrieve all articles, type:

```
$c = new Criteria();
$articles = ArticlePeer::doSelect($c);
```

The class method `doSelect()` applies the selection filter received as a parameter. In this example, the `$c` object has no rules so the `doSelect()` call will return all the records of the `Article` table (it's equivalent to a '`SELECT *`' in SQL). The variable `$articles` will contain an array of all the objects of class `Article`. To access them one by one, you can simply use a `foreach` statement:

```
foreach ($articles as $article)
{
  echo $article->getTitle();
}
```

But there is more to requests than getting all records or one record by its key. The power of the `Criteria` object is that you can add rules to it, just like you would add '`WHERE`' or '`ORDER BY`' statements in a SQL query.

To get all comments written by Steve, ordered by date, use:

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);
```

Or, because you are upset about his tone, you may want to retrieve only the comments not written by Steve:

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve', Criteria::NOT_EQUAL);
$c->addAscendingOrderByColumn(CommentPeer::DATE);
```

```
$comments = CommentPeer::doSelect($c);
```

Note the class constants name used here: these are the `phpNames` of the columns in the object data model description. The `::doSelect()` method will generate the best SQL query according to your database type, execute it, and return a list of objects - which are much better than record sets.

> **Note**: Why use `CommentPeer::AUTHOR` instead of `comment.AUTHOR`, which is the way it will be output in the SQL query? Imagine that you have to change the name of the `author` field to `contributor` in the database. If you used `comment.AUTHOR`, you would have to change it in every call to the Propel objects in your code. By using `CommentPeer::AUTHOR`, you can simply change the column name in the `schema.yml`, and keep the `phpName` `AUTHOR` to prevent multiple modifications.

Refer to the Propel Documentation for more details about the `Criteria` object and its methods.

`created_at` and `updated_at` columns

Usually, when a table has a `created_at` column, it is to store in this column a timestamp of the date when the record was created. The same applies to `updated_at` columns, that are usually updated each time the record itself is updated, to the value of the current time.

The good new is, symfony will recognize the name of these columns, and handle their update for you. You don't need to manually set the `created_at` and `updated_at` columns, they will automatically be updated.

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->save();
// show the creation date
echo $comment->getCreatedAt();
```

Additionally, the getters for these columns accept a date format as an argument:

```
echo $comment->getCreatedAt('Y-m-d');
```

Database access configuration

The data model is independent from the database used, but you will definitely use a database. The minimum information required by symfony to handle requests to the project database is the name, the access codes and the type of the database. This data should be entered in the `databases.yml` file located in the `myproject/config/` directory:

```
prod:
  propel:
    param:
      host:               mydataserver
      username:           myusername
      password:           xxxxxxxxxx

all:
  propel:
```

```
    class:                sfPropelDatabase
    param:
      phptype:            mysql
      host:               localhost
      database:           blog
      username:           root
      password:
      compat_assoc_lower: true
      # datasource:       propel
```

The database access is environment dependant. You can define different settings for the `prod`, `dev`, `test` environments, or any other environment you defined. The `all` header defines settings for all environments. This configuration can also be overridden per application, by setting different values in an application-specific file, for instance in `myproject/apps/myapp/config/databases.yml`.

> **Note:** The `host` parameter can also be defined as `hostspec`.

For each environment, you can define many connections, each being linked to a `schema.yml`. Connections and schema are linked by the `datasource` parameter. It refers to the first key in the `schema.yml`. If you don't specify the `datasource` param, it takes the value of the label given to the connection settings.

The permitted values of the `phptype` parameter are the ones of the database systems supported by Propel:

- `mysql`
- `sqlserver`
- `pgsql`
- `sqlite`
- `oracle`

The `host`, `database`, `username` and `password` settings define the parameters required to connect to the database.

In the above example, the settings for `all` environments can also be written using the shorthand syntax:

```
all:
  propel:
    class:          sfPropelDatabase
    param:
      dsn:          mysql://root:@localhost/blog
```

> **Note**: If you use a SQLite database, the `host` parameter has to be set to the path to the database file. For instance, if you keep your blog database in `myproject/data/blog.db`, the `databases.yml` will look like:
>
> ```
> all:
>   propel:
>     class:          sfPropelDatabase
>     param:
>       dsn:          sqlite://./../data/blog.db
> ```

Extended schema syntax

Attributes

Database and tables can have specific attributes. They are set under an `_attribute:` key.

You may wish that your schema gets validated before code generation takes place. To do that, deactivate the `noXSD` attribute:

```
propel:
  _attributes:   { noXsd: false }
```

The main element supports the defaultIdMethod`attribute; if none is provided then the databases native method of generating IDs will be used -- e.g. autoincrement for MySQL, sequences for PostgreSQL. The other possible value is`none`:

```
propel:
  _attributes:   { defaultIdMethod: none }
```

You already saw the `phpName` table attribute, used to set the name of the generated class mapping the table:

```
propel:
  blog_article:
    _attributes: { phpName: Article }
```

Tables that contain localized content (i.e., several versions of the content, in a related table, for internationalization) also take two additional attributes (see the internationalization chapter for details):

```
propel:
  blog_article:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
```

Column details

The basic syntax gives you two choices: let symfony deduce the column characteristics from its name (by giving an empty value) or define the type with one of the type keywords:

```
propel:
  blog_article:
    id:                # let symfony do the work
    title: varchar(50)  # specify the type yourself
```

But you can define much more for a column, and in this case you will need to define column settings as an associative array:

```
propel:
  blog_article:
    id:       { type: integer, required: true, primaryKey: true, autoincrement: true }
    name:     { type: varchar , size: 50, default: foobar, index: true }
    group_id: { type: integer, foreignTable: db_group, foreignReference: id, onDelete: cascade }
```

The column parameters are:

- `type`: Propel culomn type. See Propel list of types for more details.
- `size`: for `VARCHAR` type columns, the size of the string. Note that a column defined as `varchar(50)` in basic syntax is defined as `{ type: varchar , size: 50 }` in extended syntax.
- `required`: `false` by default, set it to `true` if you want the column to be required
- `default`: default value
- `primaryKey`: boolean, to be set to `true` for primary keys
- `autoincrement`: boolean, to be set to `true` for columns of type `integer` that need to be auto-increment
- `sequence`: sequence name for databases using sequences for auto-increment columns (e.g. PostgreSQL or Oracle)
- `index`: boolean, to be set to `true` if you want a simple index or to `unique` if you want a unique index to be created on the column
- `foreignTable`: to create a foreign key to another table.
- `foreignReference`: the name of the related column if a foreign key is defined via `foreignTable`
- `onDelete`: if set to `cascade` for a foreign key, records in this table are deleted when a related record in the foreign table is deleted.
- `isCulture`: to be set to `true` for culture columns in localized content tables (see the internationalization chapter)

Foreign key

As an alternative to the `foreignTable` and `foreignReference` column attributes, you can add manually one or many foreign keys under the `_foreign_keys` key in a table:

```
propel:
  blog_article:
    id:
    title:   varchar(50)
    user_id: { type: integer }
    _foreign_keys:
      -
        foreign_table: blog_user
        on_delete:     cascade
        references:
          - { local: user_id, foreign: id }
```

This will create a foreign key on the `user_id` column, matching the `id` column in the `blog_user` table.

As compared to the column attributes, it is interesting for multiple-references foreign keys and to give foreign keys a name:

```
   _foreign_keys:
      my_foreign_key:
        foreign_table: db_user
        onDelete:      cascade
        references:
          - { local: user_id, foreign: id }
          - { local: post_id, foreign: id }
```

Index

As an alternative to the `index` column attribute, you can add manually one or many indexes under the `_indexes` key in a table. If you want to define unique indexes, you have to use the `_uniques` header instead:

```
propel:
  blog_article:
    id:
    title:            varchar(50)
    created_at:
    _indexes:
      my_index:       [title, user_id]
    _uniques:
      my_other_index: [created_at]
```

As for the foreign-key explicit syntax, it is only useful for indexes built on more than one column.

Column automatisms

When meeting a column with no value, symfony will do some magic and add a value of its own:

```
id:         { type: integer, required: true, primaryKey: true, autoincrement: true }
foobar_id:  { type: integer, foreignTable: db_foobar, foreignReference: id }
created_at: { type: timestamp }
updated_at: { type: timestamp }
```

For the foreign key automatism, symfony will look for a table having the same `phpName` as the beginning of the column name, and if one is found, it will take this table name as the `foreignTable`.

Table automatisms (I18n)

Symfony supports content localization in related tables. This means that when you have content subject to localization, it is stored in two separated tables: one with the invariable columns, and another one with the localized columns.

In a `schema.yml`, all that is implied when you name a table `foobar_i18n`, as follows:

```
propel:
  db_group:
    id:          -
    created_at:  -

  db_group_i18n:
    name:        varchar(50)
```

Symfony will automatically add the columns and table attributes to make the localized content mechanism work. This means that the previous schema is equivalent to:

```
propel:
  db_group:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
```

```
    id:          -
    created_at:  -

  db_group_i18n:
    id:          { type: integer, required: true, primaryKey: true, foreignTable: db_group, forei
    culture:     { isCulture: true, type: varchar, size: 7, required: true, primaryKey: true }
    name:        varchar(50)
```

Beyond the `schema.yml`: The `schema.xml`

As a matter of fact, the `schema.yml` format is internal to symfony. When you call a `propel-` command, symfony actually translates this file into a `generated-schema.xml` file, which is the type of file expected by Propel to actually perform tasks on the model.

The `schema.xml` files contain the same information as their YAML equivalent. This is what the Article/Comment schema defined previously looks like in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
 <database name="propel" defaultIdMethod="native" noxsd="true">
    <table name="blog_article" phpName="Article">
      <column name="id" type="integer" required="true" primaryKey="true" autoIncrement="true" />
      <column name="title" type="varchar" size="255" />
      <column name="content" type="longvarchar" />
      <column name="created_at" type="timestamp" />
    </table>
    <table name="blog_comment" phpName="Comment">
      <column name="id" type="integer" required="true" primaryKey="true" autoIncrement="true" />
      <column name="article_id" type="integer" />
      <foreign-key foreignTable="blog_article">
        <reference local="article_id" foreign="id"/>
      </foreign-key>
      <column name="author" type="varchar" size="255" />
      <column name="content" type="longvarchar" />
      <column name="created_at" type="timestamp" />
    </table>
 </database>
```

The description of the `schema.xml` format can be found in the documentation and the getting started sections of the Propel project website.

The YAML format was designed to keep the schemas simple to read and write, but the trade-off is that the more complex schemas can't be described with a `schema.yml`. On the other hand, the XML format allows for full schema description, whatever its complexity, and includes database vendor specific settings, table inheritance, etc.

Symfony actually understands schemas written in XML format. So if your schema is too complex for the YAML syntax, if you have an existing XML schema, or if you are already familiar with the Propel `schema.xml` syntax, you don't have to switch to the YAML syntax. Place your `schema.xml` in the project `config/` directory, build the model, and there you go.

Don't create the model twice

Building a SQL database structure based on an existing schema

If you start your application by writing the `schema.yml`, you might not want to do it a second time in SQL to actually create the tables in the database. Symfony can generate a SQL query that creates the tables directly from the YAML data model. To get it, go to your root project directory and type:

```
$ symfony propel-build-sql
```

A `schema.sql` will be created in `myproject/data/sql/`. Note that the generated SQL code will be optimized for the database system defined in the `phptype` parameter of the `databases.yml` file. You can also precise the connection that you want to use in the `propel.ini`.

You can use the `schema.sql` directly to build the tables. For instance, in MySQL:

```
$ mysqladmin -u root -p create blog
$ mysql -u root -p blog < data/sql/schema.sql
```

This command is also helpful to rebuild the database in another environment, or to change RDBMS.

> **Note**: If the connection settings are properly defined in your `propel.ini`, you can even use the `symfony propel-insert-sql` command to do this automatically.

Generating an YAML data model from an existing database

Symfony can use the Creole database access layer to generate a `schema.yml` from an existing database, thanks to introspection. This can be particularly useful when you do reverse engineering, or if you prefer working on the database before working on the object model.

In order to do this, you have to make sure that the project `propel.ini` points to the correct database and contains all connection settings.

You can call the `propel-build-schema` command:

```
$ symfony propel-build-schema
```

And a brand new `schema.yml` built from your database structure is written in your `myproject/config/` directory. You can build your model based on this structure.

The schema generation command is quite powerfull and can add to your schema a lot of database dependent information. As the YAML format doesn't handle this kind of vendor information, you need to generate a XML schema to take advantage of it. It is simply done by adding an argument to the `build-schema` task:

```
$ symfony propel-build-schema xml
```

Instead of generating a `schema.yml` file, this will create a `schema.xml` fully compatible with Propel, containing all the vendor infomation. Beware that generated XML schemas tend to be quite verbose and

difficult to read.

Multiple databases

As a matter of fact, you can setup several database connections for one project, to be able to dispatch your data into several databases. To that extent, you can write several `schema.yml` files (prefix them with a special name, but keep the `schema.yml` at the end so that they are recognized).

For instance, if you want to separate the connections for the `blog` and `image repository` parts of your website, create two files in your project `config/` directory:

```
// in blog_schema.yml
blog:
  table1:
    column1: ...
    column2: ...

//in image_schema.yml
imagerep:
  table1:
    column1: ...
    column2: ...
```

Refactoring to the Data layer

When developing a symfony project, you often start by writing the domain logic code in the actions. As the projects is getting more complex, the actions contain a lot of PHP and SQL code, and become less readable.

At that point, all the logic related to the data should be moved to the Model layer. Whenever you need to do the same request in more than one place in your actions, think about transferring the related code to the Model. It helps to keep the actions small and readable.

For example, imagine the code needed in a weblog to retrieve the 10 most popular articles for a given tag (passed as request parameter). This code should not be in an action, but in the Model. As a matter of fact, if you need to display this list in a template, the action should simply look like:

```
public function executeShowPopularArticlesForTag()
{
  $tag = TagPeer::retrieveByName($this->getRequestParameter('tag'));
  $this->articles = $tag->getPopularArticles(10);
}
```

The action creates an object of class `Tag` from the request parameter. Then, all the code needed to query the database is located in a `->getPopularArticles()` method of this class. Putting the code of this method in the action would make the code much less readable.

Moving code to a more appropriate location is one of the techniques of refactoring. If you do it often, your code will be easy to maintain and to understand by other developers. A good rule of thumb about when to do refactoring to the data layer is that the code of an action should rarely count more than ten lines of PHP code.

Propel in symfony

All the details given above are not specific to symfony. As a matter of fact, you are not obliged to use Propel as your object/relational abstraction layer. But symfony works more seamlessly with Propel, since:

- All the object data model classes and the `Criteria` classes are auto-loading classes. As soon as you will use them, symfony will include the right files, but you don't need to manually include the file inclusion.
- In symfony, Propel doesn't need to be launched nor initialized. When an object uses Propel, the library initiates by itself.
- Results of a request can be easily paginated (see more in the pager chapter).
- Propel objects allow rapid prototyping and generation of a backend for your application (see more in the scaffolding chapter).
- The `schema.xml` is faster to write through the `schema.yml`.

And, as Propel is independent of the database used, so is symfony.

Symfony Controllers: actions and the front controller

Overview

The implementation of the MVC Controller paradigm in symfony uses a front controller and a set of actions. Naming conventions and fallback rules allow the PHP code of the Actions to remain clean and readable.

Front controller

All web requests are handled by a single **front controller**, which is the unique entry point to the whole application in a given environment. The default front controller, called `index.php` and located in the `myproject/web/` directory, is a simple PHP file:

```
<?php

define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/..'));
define('SF_APP',         '##APP_NAME##');
define('SF_ENVIRONMENT', 'prod');
define('SF_DEBUG',       false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATO

sfContext::getInstance()->getController()->dispatch();

?>
```

There is one front controller by environment. As a matter of fact, it is the very existence of a front controller that defines an environment. The name of the environment is set in the `SF_ENVIRONMENT` constant, after setting the root path of the application and its name.

The front controller requires the code of the `config.php` file, and this file includes all the necessary compiled configuration to allow symfony to run.

The call to the `->dispatch()` method of the Controller object does several things:

- it determines the action to execute
- if the action does not exist, it redirects to the 404 error action as defined by default
- it activates any filters for the request (for instance if the request needs authentication)
- it then executes the selected action

  Note: for your information, the singleton `sfContext::getInstance()` stores the main elements of the request:

  - the controller object (given by the method `->getController()`)
  - the request object (given by the method `->getRequest()`)
  - the logger object (given by the method `->getLogger()`)
  - ...

You will see a little further how configuration files and routing allow the Front Controller to handle any request format. But to keep it simple, the front controller basically knows how to read an URL like:

```
http://myapp.example.com/index.php/mymodule/myaction
```

And it transforms this into a call to the `myaction` action of the `mymodule` module. And the action will handle the request.

Actions

The **actions** are the heart of an application, because they contain all the application's logic, they use the Model and define variables for the View. When you make a web request in a symfony application, the URL points to an action.

Actions are located inside **module** directories. And if, to create applications and modules, you just needed to use the `symfony` command, to create actions you have to code them in PHP yourself.

Actions are defined in files called `actions.class.php` found in the `actions` subdirectory of each module directory.

For example, let's say that you created a module called `mymodule` in your `myapp` application using the appropriate `symfony` commands. The default file `actions.class.php` located in the `apps/myapp/modules/mymodule/actions/` directory defines a single action called `index`, and it looks like that:

```
class mymoduleActions extends sfActions
{
  public function executeIndex()
  {

  }
}
```

> **Note**: If you look at an actual `actions.class.php` file, you will find more than these few lines, including a lot of comments. This is because symfony recommends to use PHP comments to document your project and prepares each class file to be compatible with the phpdoc tool.

In order to call an action you need to pass the parameter 'module/action' to the front controller:

```
http://myapp.example.com/index.php/module_name/action_name
```

> **Note**: Symfony contains a **routing** mechanism that allows you to have a complete separation between the actual parameters of an action call and the form of the URL. For instance, the call to the action `index` of a module called `article`, with the parameter `id`, which is usually written:
>
> ```
> http://myapp.example.com/index.php/article/index/id/132
> ```

can be written in a completely different way with a simple change in the `routing.yml` configuration file:

```
http://myapp.example.com/articles/europe/france/finance.html
```

You will learn more about this feature in the routing chapter.

So let's try to call the `index` action:

```
http://myapp.example.com/index.php/mymodule/index
```

This new action seems to do absolutely nothing. But it can easily communicate with the view using public variables:

```
class testActions extends sfActions
{
  public function executeIndex()
  {
    $this->mytext = 'Text example';
  }
}
```

In this example, the `Index` action gives the view the opportunity to use a variable called `mytext`. In the view (the result of the action), the variable will be called like this:

```
Text: <?php echo $mytext ?>
```

But you may wonder how the action can choose the view to display its content, since it apparently does nothing. The answer is in the Action return. And, for your information, the action in this example actually did something: it called the default view.

Action return

Various behaviors are possible at the end of the execution of an action:

- if there is a default view to call (this is the most common case):

  ```
  return sfView::SUCCESS;
  ```
- if there is an error view to call:

  ```
  return sfView::ERROR;
  ```
- if there is a non-default view to call:

  ```
  return 'MyResult';
  ```
- if there is no view to call, for instance in the case of an action executed in a batch process:

  ```
  return sfView::NONE;
  ```
- for escaping view rendering but send the header content (specially X-JSON header), you can use:

  ```
  return sfView::HEADER_ONLY;
  ```
- if there is no view to call, but a value to return:

```
return sfView::VAR;
```

The returned value can be read with `$controller->getActionStock()->getFirstEntry()->getPresentation()`. This is mostly useful for unit testing (see more in the chapter dealing with the sfTestBrowser)

- if the action forwards the call to another action:

```
$this->forward('otherModule', 'index');
```

- if the action results in a web redirection:

```
$this->redirect('/otherModule/index');
$this->redirect('http://www.google.com/');
```

Note: the code located after a `forward` or a `redirect` in an action is never executed. You can consider that these calls are equivalent to a `return` statement.

Because calling the default view is the default behavior, if an action returns nothing, the framework considers that the success view has to be called as if the `return sfView::SUCCESS;` was implied.

Naming conventions

The actions are grouped in a logical way inside modules, in a directory structure such as `myproject/apps/MYAPP/modules/MODULE_NAME/actions/`, in files named `actions.class.php`. The function implementing an action in the `actions.class.php` file of the module is named by combining the word `execute` with the name of the action it refers to.

For instance, if you have two actions `index` and `list` in a module called `product`, the file `actions.class.php` located in `myapp/modules/product/actions/` will have the following structure:

```
class productActions extends sfActions
{
  public function executeIndex()
  {
    ...
  }
  public function executeList()
  {
    ...
  }
}
```

If your module has several actions and if the size of the code of an action grows so much that you need to keep it separated from the other actions of the module, you can write it in a new file, still in the same directory. You could then have an equivalent of the previous action file with two files:

- file `myapp/modules/product/actions/indexAction.class.php`:

```
class indexAction extends sfAction
{
  public function execute()
  {
```

```
    ...
  }
}
```

- file `myapp/modules/product/actions/listAction.class.php`:

```
class listAction extends sfAction
{
  public function execute()
  {
    ...
  }
}
```

Note: in this case, the action class inherits the class `sfAction` instead of `sfActions`, and the name of the action doesn't need to be repeated in the name of the `execute` function.

If you need to repeat several statements in each action before starting, or if you have similar slot inclusions for all (see below for the definition of *slots*), you should probably extract them into the `preExecute()` method of your action class:

```
class productActions extends sfActions
{
  public function preExecute()
  {
    //The code inserted here is executed at the beginning of each action call
    ...
  }
  public function executeIndex()
  {
    ...
  }
  public function executeList()
  {
    ...
  }
}
```

Guess how to repeat statements *after* every action is executed ? Wrap them in a `postExecute()` method.

Redirect or Forward

The choice between a `redirect` or a `forward` is sometimes tricky. To choose the best solution, keep in mind that a `forward` is internal to the application and transparent for the user. As far as the user is concerned, the displayed URL is the same as the one requested. On the contrary, a `redirect` is a message to the user's browser, involving a new request from it and a change in the final resulting URL.

If the action was called from a submitted form with `action="post"`, you should always do a `redirect`. The main advantage is that if the user refreshes the resulting page, the form will not be submitted again; in addition, the 'back' button works as expected.

Symfony View: templates, layouts, partials and components

Overview

The implementation of the MVC View paradigm in symfony uses templates, possibly included in a Layout. Reusable parts of code are available in slots, when they result of an action, or in fragments, when they don't need applicative logic. The naming conventions and the tight integration with the symfony Controller implementation make template management an easy task.

Templating

The result of an action execution is a View. In symfony, a View is the combination of a **template** described by a classic PHP file, and a configuration file describing the way this template will fit with other interface elements.

Here is a typical template, typically named `indexSuccess.php`:

```
<h1>Welcome</h1>
<p>Welcome back, <?php echo $name ?> !</p>
<ul>What would you like to do ?
  <li><?php echo link_to('Read the last articles', 'article/read/') ?></li>
  <li><?php echo link_to('Start writing a new one', 'article/write/') ?></li>
</ul>
```

It contains some HTML code and some basic PHP code, usually calls to variables defined in the action and helpers.

The variables called in the templates must be either one of the usual shortcuts (see below) or attributes of the action object defined in the related action file. For instance, to define a value for the `$name` variable used here, the action must contain such a line:

```
$this->name = 'myvalue';
```

Developers like to keep as little PHP code as possible in templates, since these files are the ones used to design the GUI of the application, and are sometimes created and maintained by another team, specialized in presentation but not in application logic.

Shortcuts

Symfony provides the templates with a few useful pre-defined variables, or **shortcuts**, allowing it to access the most commonly needed information:

```
$sf_context        //the whole context object
$sf_request        //the origin request
$sf_params         //parameters of the origin request
$sf_user           //the current sfUser object
$sf_view           //the calling sfView object
```

For instance, if the action call included a parameter `total`, the value of the parameter is available in the template with:

```
<?php echo $sf_params->get('total'); ?>
// equivalent to the following action code:
echo $this->getRequestParameter('total');
```

This is true whether the parameter was sent to the action with POST or GET methods, or using the symfony routing system:

```
index.php/test/index&total=123   //GET
index.php/test/index/total/123   //routed action call
```

And since the first action call can lead to other action calls (for instance if there is a `forward`), you may need to access the action stack. A few additional shortcuts will make it easy for you:

```
$sf_first_module   //first module called
$sf_last_module    //last module called
$sf_first_action   //first action called
$sf_last_action    //last action called
```

Helpers

Helpers (like the `link_to()` function in the template example above) are PHP functions that facilitate the process of writing templates and produce the best possible HTML code in terms of performance and accessibility. Three types of helpers are available:

- The standard compulsory helpers, that must be used instead of the corresponding HTML code because they allow internal symfony mechanisms (like routing, automated form management, internationalization, etc.) to work.

  This includes all the HTML tags that handle URLs: they are replaced by symfony shortcuts to allow the routing.
- The standard optional helpers, that use less code than classic HTML for the same purpose.

  In addition, they take advantage of the symfony architecture : by using these helpers, you make sure that your code will even work after any subsequent change in the file tree structure.
- The helpers defined specifically for an application (learn more about that feature in the custom helper creation chapter).

Using helpers speeds up the template development; they are described in detail in the following chapters.

In order to use helper functions, the file that contains this helper has to be loaded.

A few helpers are loaded for every application:

- Helper: defines the `use_helper()` helpers, needed for helper inclusion
- Tag: defines the basic tag operations
- Url: link and URL management helpers
- Asset: head, include, images and javascript call helpers

By default, the application configuration specifies additional helpers to be loaded for all requests in the `settings.yml`:

```
default:
  .settings:
    standard_helpers:    [Partial, Cache, Form]
```

You can define the helpers that need to be included in your application in addition to those ones. For instance, if you know that your application will use a lot of text and internationalization functions, you can write in the `settings.yml`:

```
all:
  .settings:
    standard_helpers:    [Partial, Cache, Form, Text, I18N]
```

If you need to use a helper that is not loaded by default, the template that will use it has to call the `use_helper()` function:

```
// use a specific in this template
<?php echo use_helper('Date') ?>
...
<?php echo format_date($date, 'd', 'en') ?>

// you can call more than one helper at once
<?php echo use_helpers('Date', 'I18N') ?>
```

Global template

All the templates can be included into a **global template**, or **layout**, located in `myproject/apps/myapp/templates/layout.php`. Here is its default content:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/2000/REC-xht
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

<?php echo include_http_metas() ?>
<?php echo include_metas() ?>

<?php echo include_title() ?>

<link rel="shortcut icon" href="/favicon.ico">

</head>
<body>

<?php echo $content ?>

</body>
</html>
```

The content of the `<head>` tag might be a little cryptic for now, but if will be explained in detail in the next chapter. Just keep in mind that with the default configuration, and our previous template, the processed view should look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/2000/REC-xht
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta name="robots" content="index, follow" />
<meta name="description" content="" />
<meta name="keywords" content="" />
<title>symfony project</title>
<link rel="stylesheet" type="text/css" href="/css/main.css" />
<link rel="shortcut icon" href="/favicon.ico">
</head>
<body>

<h1>Welcome</h1>
<p>Welcome back, <?php echo $name ?> !</p>
<ul>What would you like to do ?
  <li><?php echo link_to('Read the last articles', 'article/read/') ?></li>
  <li><?php echo link_to('Start writing a new one', 'article/write/') ?></li>
</ul>

</body>
</html>
```

This is an application of the decorator design pattern: The content of the template called by the action is integrated into the global template with the help of the `$content` variable.

The global template can be entirely customized for each application.

Naming conventions

Now you may wonder how symfony links actions with templates. To avoid repeated explicit call of templates when the name of the template can be easily deduced from the one of the action, symfony has a few naming conventions.

A template name is made of two parts: the first part is related to the action, the second to is result. For instance, the typical template of an action called `index` that executed successfully is `indexSuccess.php`.

This is an implicit rule in symfony: if the value returned by the action is `sfView::SUCCESS`, then the name of the template called will be the name of the action concatenated with `Success.php`.

For instance, with two actions `index` and `list` in a `product` module :

```
class productActions extends sfActions
{
  public function executeIndex()
  {
    ...
    return sfView::SUCCESS;
  }
  public function executeList()
  {
    ...
    return sfView::SUCCESS;
  }
```

```
}
```

The templates called at the end of the execution of each action will be located in `myproject/apps/myapp/modules/product/templates/` and named:

```
indexSuccess.php
listSuccess.php
```

Because this is the default behavior, if an action returns nothing, the framework considers that the success view has to be called as if the `return sfView::SUCCESS;` was implied.

For any other return value (different from `sfView::NONE`), the template called will have a name ending with this value. For instance, if the action returns `sfView::ERROR`, the template name must end with `Error.php`.

An action can also set an alternate template:

```
$this->setTemplate('myCustomTemplate');
```

According to the value returned by the action, the template called can be `myCustomTemplateSuccess.php` or `myCustomTemplateError.php`.

Now let's see all these possibilities in action:

```
class productActions extends sfActions
{
  public function executeIndex()
  {
    if ($test == 1)
    {
      ...
      return sfView::SUCCESS;
    }
    else if ($test == 2)
    {
      ...
      return 'MyResult';
    }
    else if ($test == 3)
    {
      ...
      $this->setTemplate('myCustomTemplate');
      return sfView::SUCCESS;
    }
    else
    {
      return sfView::ERROR;
    }
  }
}
```

The templates called in `myapp/modules/product/templates/`, according to the value of `$test`, are named:

```
indexSuccess.php
indexMyResult.php
myCustomTemplateSuccess.php
indexError.php
```

View configuration

Applications and modules can have a `view.yml` configuration file describing how the templates are integrated with other interface components (layouts, slots, stylesheets, etc.) in the module and each of its actions.

So the `view.yml` file is an alternative solution to the function calls concerning the view from within the action:

```
template:     $this->setTemplate();
```

Read more about the `view.yml` files in the next chapter.

Code fragments

You may often need to include some HTML or PHP code into several pages, to avoid repeating it, or to display the same piece of content throughout more that one page. The PHP `include()` statement will suffice most of the time for that purpose. But what if you need to access the symfony objects and helpers, or pass parameters to this code fragment? Symfony provides three helpers for that:

- If the logic is lightweight, you will just want to include a template file having access to some data you pass to it. For that, you will use a **partial**.
- If the logic is heavier (for instance if you need to access the data model and/or modify the content according to the session), you will prefer to separate the presentation from the logic. For that, you will use a **component**
- If the nature of the fragment depends on the context (for instance if the fragment needs to be different for the actions of a given module), you will use a **component slot**.

In addition, symfony provides a technique to define a zone in the layout that can be overridden by a template, and it's called a **slot**. It is an alternative to component slots for when you have no heavy logic.

These helpers are available from any symfony template.

Include

If the content type is static HTML, you can use a classic `include()` PHP statement:

```
// include some HTML code into the current template
<?php include('myfile.html') ?>
```

The most common need is to include a piece of PHP code. If this code doesn't require the symfony objects and methods, the `include()` statement is still enough. For instance, if many of the templates of your `test` module use a fragment of code showing the current time, save this piece of code in a file called `time.php` in

the global template directory (`myproject/apps/myapp/templates`). Now, when you need this piece of code in a template, just call the `include()` function:

```
// include some PHP code into the current template
<?php include(sfConfig::get('sf_app_template_dir').'/time.php') ?>
```

Partial

Partials are template fragments that can be called from different modules throughout an application. A partial has access to the usual symfony helpers and template shorcuts, but not to the variables defined in the action calling it, unless passed explicitly as an argument.

Let's imagine an application where the user has a selection of items in a shopping cart. In some pages of the application, a block displays the number of items in the cart and a link to it. The shopping cart object is provided by the action. First, let's see what the template partial would look like:

```
<div>
  <h1>Selected items</h1>
  <?php echo $cart->getCount() ?><br />
  <?php echo link_to('Modify', 'cart/edit?cart_id='.$cart->getId()) ?>
</div>
```

This template partial is saved in a `cart` module, in a file called `templates/_sidebar.php`.

Now, let's go back to the templates using this partial. For each of them, the action defines a `$myShoppingCart` object containing the current selection of items. To pass this value to a partial, the templates will use the `include_partial()` helper:

```
<?php echo include_partial('cart/sidebar', array('cart' => $myShoppingCart )) ?>
```

The first argument is the module/template name, the second is an array of variables that are needed by the partial. Notice the '_' difference between partial name in the `include_partial()` call and the actual file name in the `templates/` directory. This helps to keep your code clean and to show the fragments ahead of the other templates in a file explorer.

If your need is not restricted to one module, save your fragment in the main template directory (in `myproject/apps/myapp/templates`). To call it from any module template, you now need:

```
<?php echo include_partial('global/sidebar', array('cart' => $myShoppingCart )) ?>
```

Components

In the previous example, the logic of the partial is light and doesn't require an action. If you need a partial with a logic behind, you should use a component. A component is like an action, it can pass variables to a template partial - except it's much faster than an action. The logic is kept in a `components.class.php` file in the `actions/` directory, and the template is a regular partial.

For instance, a news component could display in a sidebar the latest news headlines for a given subject, depending on the user's profile. The queries necessary to get the news headlines are too complex to appear in a simple partial, so they have to be moved to an action-like file.

In a `news` module, create a file called `components.class.php` in the `actions/` folder with the following content:

```
<?php

class newsComponents extends sfComponents
{
  public function executeHeadlines()
  {
    $c = new Criteria();
    $c->addDescendingOrderByColumn(NewsPeer::PUBLISHED_AT);
    $c->setLimit(5);
    $this->news = NewsPeer::doSelect($c);
  }
}

?>
```

In the same `news` module, create a `_headlines.php` partial in the `templates/` directory:

```
<div>
  <h1>Latest news</h1>
  <ul>
  <?php foreach($news as $headline): ?>
    <li>
      <?php echo $headline->getPublishedAt() ?>
      <?php echo link_to($headline->getTitle(), 'news/show?id='.$headline->getId()) ?>
    </li>
  <?php endforeach ?>
  </ul>
</div>
```

Now, every time you need the component in a template, just call it by:

```
<?php include_component('news', 'headlines') ?>
```

Just like the partials, components accept additional parameters in the shape of an associative array. The parameters are available to the partial under their name, and in the component via the `$this` object:

```
// call to the component
<?php include_component('news', 'headlines', array('foo' => 'bar')) ?>

// in the component itself,
echo $this->foo;          => 'bar'

// in the _headlines.php partial,
echo $foo;                => 'bar'
```

You can include components in components, or in the global layout, as in any regular template. Like actions, components `execute` methods can pass variables to the related partial and have access to the same shortcuts. But the similarities stop there: A partial doesn't handle security, doesn't have various `return` possibilities, and is much faster than an action to execute.

Component slots

You sometimes need to include a component which varies according to the module calling it. For instance, the main layout of an application can display in the right part of the window a set of contextual information and links.

Symfony makes it easy to handle with the `include_component_slot()` helper. This function expects a label as parameter, and this label will be used to define a component for each module, using the `view.yml` configuration file.

For instance, let's imagine that the `layout.php` of the application contains:

```
...
<div id="sidebar">
  <?php include_component_slot('sidebar') ?>
</div>
```

In the main `view.yml` configuration file (located in the `myapp/config/` directory), you define the default value for this slot:

```
default:
  components:
    sidebar:  [bar, default]
```

By default, this sidebar will call the `executeDefault()` method of the `barComponents` class located in the `bar` module, and this method will display the `_default.php` partial located in `modules/bar/templates/`. But you can override this setting for a given module. For instance, in a `user` module, you may want the contextual component to display the user name, and the number of articles he/she published. In that case, add to the `view.yml` of the `modules/user/config/` directory:

```
all:
  components:
    sidebar:  [bar, user]
```

> **Note**: The `config/view.yml` file for your module is not automatically generated by the module initialization. You will need to create this file manually.

In the `modules/bar/actions/components.class.php`, add this new method:

```
class barComponents extends sfComponents
{
  ...
  public function executeUser()
  {
    $current_user = $this->getUser()->getCurrentUser();
    $c = new Criteria();
    $c->add(ArticlePeer::AUTHOR_ID, $current_user->getId());
    $this->nb_articles = ArticlePeer::doCount($c);
    $this->current_user = $current_user;
  }
}
```

Then, create the following `modules/bar/templates/_user.php`:

```
User name: <?php echo $current_user->getName() ?><br />
(already published <?php echo $nb_articles ?> articles)
```

Component slots can be used for breadcrumbs, contextual navigations, and dynamic insertions of all kinds. As components, they can be used in the global layout and in regular templates, or even in other components. The configuration setting the component of a slot is always the one of the last action called.

> **Note**: If you need to suspend the use of a component slot for a given module, just declare an empty module/component for it:
>
> ```
> all:
>   components:
>     sidebar:  [ ]
> ```

Slots

Managing a component slot obliges you to create a component, a partial, and a configuration file. It is really useful when you want reusability. If your need is to define an additional zone in the layout that can be overriden action by action, then there is a faster solution: the slots.

A slot is a code chunk stored globally in the response. It can be defined anywhere (in the layout, in a template, in a partial) and can be included anywhere as well. Just remember that the layout is executed after the template, and that the partials are executed when they are called in a template, to be sure to define a slot before including it.

To define a slot, use the `slot()` and `end_slot()` helpers. For instance, a template will define the content of a 'sidebar' slot as follows:

```
...
<?php slot('sidebar') ?>
  <h1>User <?php echo $user ?>'s details</h1>
  <p>email: <?php echo $user->getEmail() ?></p>
  <p><?php echo link_to('do things', 'user/dothings?id='.$user->getId()) ?>
<?php end_slot() ?>
```

The code between the `slot()` and `end_slot()` helpers is not output in the template, but in a placeholder in the response, waiting to be included somewhere else.

To include a slot, use the `include_slot()` helper. Using the `has_slot()` helper, you can even manage default slot values. For instance, the layout can display the 'sidebar' slot in a `<div>`. If the slot has been defined earlier (by a template, as in the previous example), then it is displayed. If not, the default content takes its place. Note that since `include_slot()` returns nothing if the slot is not defined, the use of `has_slot()` is not compulsory.

```
...
<div id="sidebar">
  <?php if(has_slot('sidebar'): ?>
    <?php include_slot('sidebar') ?>
```

```
  <?php else: ?>
    <!-- default sidebar code -->
    <h1>Contextual zone</h1>
    <p>This zone contains links and information relative to the main content of the page.</p>
  <?php endif; ?>
</div>
```

One common usage of slots is the addition of special tags in the `<head>` section, for instance when you want to include RSS links in the header for some specific actions. A layout with more than one dynamic zone, for instance the main content plus the secondary navigation, is another commun use. As they can be included anywhere, with fallback mechanisms, slots are very powerful and allow for a very modular conception of layouts and templates. Eventually, slots are much faster than component slots, so feel free to use them extensively.

> **Note**: Slots were present in earlier versions of symfony, but with a slightly different meaning.

View Configuration

Overview

The View is made of:

- one part resulting of the action execution (the template)
- all the rest (headers, metas, file inclusions, slots and layout)

Symfony offers two convenient ways to modify the components of the View that are not part of the template.

The two ways to modify the view

Apart from the template itself, which integrates the variables calculated in the action into the view, every action needs control over other parts of the view:

- **meta declarations**: keywords, description or cache duration
- **page title**: not only does it help users with several browser windows open to find yours, but it is also very important for search sites indexing
- **file inclusions**: javascript and stylesheet files
- **slots inclusions**: navigation bar, header, footer, breadcrumb, etc.
- **layout**: some actions require a custom layout (popups, ads, ajax, etc.)

Ideally, all these parameters should be defined in a configuration file, so that the action can deal only with application logic and leave presentation to the view. However, in many cases, these parameters have dynamic values (for instance, page titles often show the title of articles) and cannot be kept in static configuration files. That's why symfony provides two ways to modify them:

- the `view.yml` file, in which these parameters can be set at the application, module and action levels
- the **view helpers**, which are methods accessible from within the action.

According to the nature of the values that you need to give to these parameters, you will prefer one method or another. It is recommended that you keep as much configuration in the `view.yml` files as possible, since it separates clearly logic from presentation.

> **Note**: the `view.yml` solution offers the advantage of inheritance and specialization. If you have to use the helpers in the action but you don't want to repeat them for every action, use the `preExecute()` and `postExecute()` methods of the module class to avoid repeating helpers for every action of a module.

View.yml structure

The scope of a `view.yml` varies according to the place where it is stored:

- in `myproject/apps/myapp/config/`, the definitions will apply to all modules and all actions of the application

- in `myproject/apps/myapp/modules/mymodule/config/`, the definitions will apply only to one module and override the application-level definitions

In addition, for the module-level files, you have the ability to specify custom definitions for a specific view (pair action/result of the action):

```
view1:
  ...
view2:
  ...
all:
  ...
```

The next sections will detail how the parameter definitions in this file can affect the view:

- http-metas: *(associative array)*
  - ♦ key: value *(string)*
- metas: *(associative array)*
  - ♦ key: value *(string)*
- title: page_name *(string)*
- stylesheets: [stylesheet1, stylesheet2, ...] *(array)*
- javascripts: [javascript1, javascript2, ...] *(array)*
- layout: layout_name *(string)*
- slots: *(associative array)*
  - ♦ slot_name: [module, action] *(array)*

Default configuration

The default global template, created when you initialize an application, is called `layout.php`. It is located in `myproject/apps/myapp/templates/` and looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/2000/REC-xht
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

<?php echo include_http_metas() ?>
<?php echo include_metas() ?>

<?php echo include_title() ?>

<link rel="shortcut icon" href="/favicon.ico">

</head>
<body>

<?php echo $content ?>

</body>
</html>
```

The default settings of the view, set in the application view configuration file `myproject/apps/myapp/config/view.yml`, are:

```
all:
  http_metas:
    content-type: text/html; charset=utf-8

  metas:
    title:        symfony project
    robots:       index, follow
    description:  symfony project
    keywords:     symfony, project

  stylesheets:    [main]

  javascripts:    [ ]

  has_layout:     on
  layout:         layout

  components:     {}
```

When applied to the layout, these settings produce the following `<head>` section:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/2000/REC-xht
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta name="title" content="symfony project" />
<meta name="robots" content="index, follow" />
<meta name="description" content="symfony project" />
<meta name="keywords" content="symfony, project" />

<title>symfony project</title>

<link rel="stylesheet" type="text/css" media="screen" href="/css/main.css" />

<link rel="shortcut icon" href="/favicon.ico">

</head>
<body>

<?php echo $content ?>

</body>
</html>
```

Besides the obvious use of the parameter definitions in the `view.yml`, this example shows that the `<head>` section of a page can be modified even if the code for this section is normally in a file shared by all modules.

View configuration could stop at that point if you wish. There is no absolute necessity to define view parameters for each module, since the default behaviors of actions will manage for you the choice of the template to be used. However, as soon as your application uses more than one layout, custom javascripts or stylesheets, or if the interface shows common components on every page, writing `view.yml` files for your modules will save you time and ease your work.

> **Note**: If you want to create custom layouts, you can use the `include_` functions to get the values of the head parameters and take advantage of the configuration files.

Meta configuration

The information written in the `meta` tags is not displayed but is useful for robots and search engines. It also controls the cache settings of every page.

To set these parameters for a modules, use these keys in a `view.yml` file:

```
http_metas:
  $key: $value

metas:
  $key: $value
```

To modify these settings from within an action, use these methods:

```
$this->getResponse()->addHttpMeta($key, $value)
$this->getResponse()->addMeta($key, $value)
```

Adding an existing key will replace its current content.

Let's image that you display an article about finance in France that will not change. You can write in the view configuration file:

```
http_metas:
  cache-control:public

metas:
  description:  Finance in France
  keywords:     finance, France
```

Or, alternatively, call in the action:

```
$this->getResponse()->addHttpMeta('cache-control', 'public');
$this->getResponse()->addMeta('description','Finance in France');
$this->getResponse()->addMeta('keywords', 'finance, France');
```

When the template is included in the layout, the `include_` helpers of the `<head>` part will give:

```
<meta http-equiv="cache-control" content="public">
<meta name="description" content="Finance in France" />
<meta name="keywords" content="finance, France" />
```

As a bonus, the HTTP header of the HTML file will also contain the `http-metas` setting. This means this if you need to modify the HTTP headers, the `http-metas` section of the `view.yml` is the solution, even if you don't have any `<?php echo include_http_metas() ?>` in the layout -- or if you have no layout. For instance, if you need to send a page as plain text, type:

```
http_metas:
  content-type: text/plain

has_layout: false
```

Title configuration

Page title is a key part to search engine indexing; it is also very useful with modern browsers that provide tabbed-browsing.

To have a custom title for every page, take advantage of the `<?php echo include_title() ?>` call in the `<head>` part of the layout.

If your title only depends on the view, you should specify `title` in the `view.yml`:

```
indexSuccess:
  metas:
    title: Description

indexError:
  metas:
    title: Problem while displaying the description

listSuccess:
  metas:
    title: Result list
```

On the other hand, if an action can result in a page with various titles, you should use the `setTitle($title)` function:

```
$this->getResponse()->setTitle($title);
```

File inclusion configuration

Basic syntax

To include a specific stylesheet or a specific javascript file in the view configuration file, use:

```
stylesheets: [$css]
javascripts: [$js]
```

To do it from an action, use:

```
$this->getResponse()->addStylesheet($css);
$this->getResponse()->addJavascript($js);
```

In each case, the argument is a file name. If it has a logical extension (`.css` for a stylesheet and `.js` for a javascript), you can omit it. If it has a logical location (`/css/` for a stylesheet and `/js/` for a javascript), you can also omit it.

For instance, in an action called `subscribe` using a calendar widget, the view could be configured with:

```
subscribeSuccess:
  stylesheets: [calendar/skins/aqua/theme]
  javascripts: [calendar/calendar_stripped, calendar/calendar-en, calendar/calendar-setup_strippe
```

Alternatively, you could require the files from the action with:

```
$this->getResponse()->addStylesheet('calendar/skins/aqua/theme');
$this->getResponse()->addJavascript('calendar/calendar_stripped');
$this->getResponse()->addJavascript('calendar/calendar-en');
$this->getResponse()->addJavascript('calendar/calendar-setup_stripped');
```

The `<head>` of the resulting view would then show:

```
<link rel="stylesheet" type="text/css" media="screen" href="/js/calendar/skins/aqua/theme.css" />
<script language="javascript" type="text/javascript" src="/js/calendar/calendar_stripped.js"></sc

<script language="javascript" type="text/javascript" src="/js/calendar/lang/calendar-en.js"></scr
<script language="javascript" type="text/javascript" src="/js/calendar/calendar-setup_stripped.js
```

> **Note**: In previous versions of symfony, this type of inclusion was only possible if you included calls to the `include_stylesheets()` and `include_javascripts()` helpers in the `<head>` section of the layout. This is not necessary anymore, and the use of these helpers is deprecated.

Addition, removal, ordering

When you mention a stylesheet or a JavaScript in a `view.yml`, it appears in the response `<head>` *in addition to the ones already defined* and after them. The addition of these files follows the usual cascade definition, and the order of inclusion is from the most general to the most particular.

For instance, if your application `view.yml` stipulates:

```
default:
  stylesheets: [main]
```

And if you have, in your module `config/view.yml`:

```
indexSuccess:
  stylesheets: [special]

all:
  stylesheets: [additional]
```

Then the `<head>` of the resulting `indexSuccess` view will show:

```
<link rel="stylesheet" type="text/css" media="screen" href="/css/main.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/additional.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/special.css" />
```

The same applies for the JavaScript inclusions. This allows you to override global settings at the module and view level.

To give you more control over this cascade, you can remove an already included stylesheet or JavaScript by prefixing its name with a minus sign ('-') in the `view.yml`

```
# adds a new stylesheet and removes the default one (as defined in the global view.yml)
```

```
indexSuccess:
  stylesheets: [special, -main]
```

To remove all stylesheets or JavaScripts, use the following syntax:

```
# remove all stylesheets and JavaScripts
indexSuccess:
  stylesheets: [-*]
  javascripts: [-*]
```

You can be more accurate from within an action and define an additional parameter to force the position where to include the file (`first` or `last` position):

```
$this->getResponse()->addStylesheet('special', 'first');
```

Media specific stylesheet

To specify a media for a stylesheet inclusion, you can change the default stylesheet tag options:

```
stylesheets: [main, main_for_print: { media: print }]
```

or

```
$this->getResponse()->addStylesheet('main_for_print', '', array('media' => 'print'));
```

Layout configuration

According to the graphical charter of your website, you may have several layouts. Classical websites have at least two: the default layout, and the popup layout.

You already saw that the default layout is `myproject/apps/myapp/templates/layout.php`. You don't need to set it in the application-level view configuration. Additional layouts have to be added in the same directory.

If you need to use a different layout for a view, use:

```
layout: $layoutname
```

Or in the action:

```
$this->setLayout($layoutname);
```

For instance, if all the actions of the `rules` module use the `content.php` layout, except the `securitypolicy` action for which you need the `popup.php` layout, you can write:

```
securitypolicySuccess:
  layout: popup

all:
  layout: content
```

The same is possible in the `Actions.class.php` file:

```
class rulesActions extends sfActions
{
  public function preExecute()
  {
    $this->setLayout('content');
  }

  public function executeSecuritypolicy()
  {
    $this->setLayout('popup');
    ...
  }
}
```

Some views don't need any layout at all (for instance plain text pages). In that case, use:

```
layout: false
```

Or, in the action:

```
$this->setLayout(false);
```

Component slots configuration

Common interface components with dynamic content (breadcrumbs, navigation bars, etc.) that require the execution of a component (the "light" version of an action) are published in *component slots* in symfony. The definition in the `view.yml` looks like:

```
components:
  $componentname: [$module, $action]
```

Imagine that you have a layout with a zone for contextual advices:

```
...
<div id="adviceBlock">
  <?php include_component_slot('advices') ?>
</div>
...
```

You also have a module `advice` with contextual components `forIndex` and `forList`. Now, if you want to associate a contextual component with the views of a module `article` with actions `index` and `list`, you could write the following `myproject/apps/myapp/modules/article/config/view.yml`:

```
indexSuccess:
  components:
    advices: [advice, forIndex]

listSuccess:
  components:
    advices: [advice, forList]
```

How about template configuration?

Though you could expect to see a way to set a custom template with one of the two ways described here, there is none.

The reason is that when you need to set a custom template for an action, you should do either a `redirect` or a `forward` of this action to another one. The context will be much cleaner and the logic of your application will be preserved.

Templating in practice : Link helpers

Overview

One of the common needs in templates is the automation of the URL generation, and the inclusion of local files regardless of the file structure. The link and image helpers, for instance, naturally replace the `<a>` and `<img>` tags and do all the tedious jobs for you. Their use is compulsory to enable routing and relative links affected by configuration.

Introduction

The way to call an action with an URL has already been explained: put the module and action names after the front controller, then the parameters of the action either in the classic "GET" style or between `/`:

```
http://myapp.example.com/index.php/article/read?title=Finance_in_France
// is equivalent to
http:/myapp.example.com/index.php/article/read/title/Finance_in_France

// additionally, the following works if the routing is deactivated
http://myapp.example.com/index.php?module=article&action=read&title=Finance_in_France
```

Imagine that you want to add a hyperlink to this page in a template. In PHP, you could be tempted to write:

```
<a href="/index.php/article/read?title=Finance_in_France">Finance in France</a>
```

This is a bad idea, because:

- by writing `index.php`, you point to a particular environment (in this case, the production environment). All links must be environment-independent if you don't want to accidentally change environment while navigating
- if you later decide to change the **routing** policy (you can learn more about this feature in the routing chapter chapter), and to display another format of URL, you would have to parse all of your template files and replace by hand the URLs already entered to URLs like:

  ```
  [php]
   <a href="/index.php/articles/europe/france/finance.html">Finance in France</a>
  ```

If you ever built a website, you know that these problems occur all the time.

The link helpers are there for you. They allow symfony to create URLs and hyperlinks to actions compliant to the routing policy, without any dependence on the environment.

Link helpers

Syntax

The `url_for()` helper returns an URL, the `link_to()` helper returns a hyperlink. You may never use the

first one, but you will use `link_to()` a lot.

```
<?php echo url_for('article/read?title=Finance_in_France') ?>
// will generate in HTML
http://myapp.example.com/index.php/article/read/title/Finance_in_France

<?php echo link_to('interesting article', 'article/read?title=Finance_in_France') ?>
// will generate in HTML
<a href="http://myapp.example.com/index.php/article/read/title/Finance_in_France" title="interest
```

Notice that the default routing rules transform all the `?`, `=` and `&` characters into `/`. Also note that the `link_to()` helper gives a bonus `title` attribute for the `<a>` tag, based on the content of the tag that is provided.

Enter in the link helper exactly the same URL as you would have written in a `<a href="...` tag, and it will transform it for you into a "smart" link according to the current routing policy. And when the customer decides to change some of the rules, just modify the `routing.yml` configuration file and the generated URL can now look like:

```
<a href="http://myapp.example.com/articles/europe/france/finance.html" title="interesting article
```

Link helpers are the key to a double way routing. Interpreting URLs is not a big deal, but generating them requires that all hyperlink tags pass by the same filter. That's why these helpers are compulsory as soon as you may use the routing feature. Make sure you take a look at the routing chapter to see how to change the shape of URLs, get rid of the controller name or add a '.html' to the generated URLs for more efficient caching and indexing.

Linking an image

Adding a link to an image is as simple as combining two helpers:

```
<?php echo link_to(image_tag('read.gif'), 'article/read?title=Finance_in_France') ?>
//will generate in html
<a href="/article/read/title/Finance_in_France.html"><img src="/images/read.gif" alt="Read" /></a
```

The `image_tag` helper is described later in this chapter.

Using the rule instead of the module/action

If you had a look at the routing chapter, you saw that the routing is defined by rules. The link helpers accept a rule name instead of a `module/action` pair if an `@` is written before. For instance, if the `routing.yml` file contains the rule:

```
cart:
  url: /shopping_cart
  param: { module: shoppingCart, action: index }
```

Then you can write

```
<?php echo link_to('go to shopping cart', '@cart') ?>
```

instead of

```
<?php echo link_to('go to shopping cart', 'shoppingCart/index') ?>
```

There are pros and cons to this trick. The advantages are:

- The routing is done much faster, since symfony doesn't have to browse all the rules to find the one that matches the link. In a page with a great number of routed hyperlinks, the boost will be noticeable if you use rules instead of `module/action`.
- Using the rule helps to abstract the logic behind an action. If the need to plug another action behind a given URL arises, then only the `routing.yml` file will need to be changed - all the `link_to()` calls will still work without intervention.
- The logic of the call is more apparent with a rule name. Even if your modules and actions have explicit names, it is often better to call "display_article_by_title" than "article/display".

On the other hand,

- If you write rules in your `link_to()`, you can not deactivate the routing anymore.
- Debugging an application becomes less self-evident since you always need to refer to the `routing.yml` file to discover which action is called by a link.

The best choice depends on the project. In the long run, it's up to you.

Fake GET and `post` option

Sometimes, web developers tend to use GET requests to actually do a POST. For instance, consider the following URL:

```
http://myapp.example.com/index.php/shopping_cart/add?id=100
```

This request will change the data contained in the application, so it should be considered as a POST.

This URL can be bookmarked, cached and indexed by search engines. Imagine all the nasty things that might happen to the users or to the statistics of a website using this technique.

Symfony provides a way to transform a call to a `link_to` helper into an actual post. Just write:

```
<?php echo link_to('go to shopping cart', 'shoppingCart/add?id=100', 'post=true') ?>
// will generate in HTML
<a onclick="f = document.createElement('form'); document.body.appendChild(f); f.method = 'POST';
```

This `<a>` tag has an `href` attribute, so browsers without javascript support will follow the link doing the default GET. But for those with javascript support, the click on the link will actually do a POST request to the same URL, thanks to a dynamically-generated form in the `onclick` behavior.

It is a good habit to tag as POST the links that actually post data. Don't hesitate to use this possibility.

Additional options

In addition to the `post` option, the `link_to` helper accepts two other options : `confirm` and `popup`.

```
<?php echo link_to('add to cart', 'shoppingCart/add?id=100', 'confirm=Are you sure?') ?>
// will generate in HTML
<a onclick="return confirm('Are you sure?');" href="/fo_dev.php/shoppingCart/add/id/100.html">add

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', 'popup=true') ?>
// will generate in HTML
<a onclick="window.open(this.href);return false;" href="/fo_dev.php/shoppingCart/add/id/100.html"

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', Array('popup' => Array('Window title
// will generate in HTML
<a onclick="window.open(this.href,'Window title','width=310,height=400,left=320,top=0');return fa
```

The options can be combined:

```
<?php echo link_to('add to cart', 'shoppingCart/add?id=100', 'confirm=Are you sure? post=true') ?
// will generate in HTML
<a onclick="if (confirm('Are you sure?')) { f = document.createElement('form'); document.body.app

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', 'confirm=Are you sure? popup=true')
// will generate in HTML
<a onclick="if (confirm('Are you sure?')) { window.open(this.href); };return false;" href="/fo_de
```

Forcing GET variables

According to your routing rules, variables passed as parameters to a `link_to()` are transformed into patterns. The default rule transforms `?key=value` into `/key/value`:

```
<?php echo link_to('interesting article', 'article/read?title=Finance_in_France') ?>
// will generate in the URL
http://myapp.example.com/index.php/article/read/title/Finance_in_France
```

But what if you actually need to keep the GET syntax, i.e. to output an url like:

```
http://myapp.example.com/index.php/article/read?title=Finance_in_France
```

You should then put the variables that have to be forced outside of the `url` parameter, in the `query_string` option:

```
<?php echo link_to('interesting article', 'article/read', array('query_string' => 'title=Finance_
```

button_to helper

When dealing with forms, you might want to use buttons instead of links. This is where the `button_to` helper enters:

```
<?php echo button_to('add to cart', 'shoppingCart/add?id=100') ?>
// will generate in HTML
<input value="add to cart" type="button" onclick="document.location.href='/fo_dev.php/shoppingCar
```

Note that, by default, this button will do a GET request. If you want to do a POST, add the related option:

```
<?php echo button_to('add to cart', 'shoppingCart/add?id=100', 'post=true') ?>
// will generate in HTML
<form method="post" class="button_to" action="/fo_dev.php/shoppingCart/add/id/100.html"><input va
```

Beware not to use the POST mode when in a form, since it adds a new `<form>` tag: you would end up with a form inside another, and the additional button would submit the first form.

As for `link_to`, the `confirm` and `popup` options can be added and combined:

```
<?php echo button_to('add to cart', 'shoppingCart/add?id=100', 'confirm=Are you sure? popup=true'
// will generate in HTML
<input value="add to cart" type="button" onclick="if (confirm('Are you sure?')) {window.open('/fo
```

mail_to helper

The reason why symfony ships a `mail_to()` helper is that nowadays, email harvesting robots prowl about the web, and you can't display an email address on a website without becoming a spam victim within days.

This helper takes three parameters : the actual email address, the string that should be displayed and an array of options. If the third parameter is not set the email address is written non-crypted:

```
<?php echo mail_to('myaddress@mydomain.com', 'contact address') ?>
// will generate in HTML
<a href="mailto:myaddress@mydomain.com'>contact address</a>
```

If the flag is set to array('encode' => true) the displayed and pointed addresses are processed by a random decimal and hexadecimal entity encoder. This will probably not stop all address-harvesting spambots, but it will keep most of them away. Future versions of this helper will probably evolve as the harvesting techniques become more accurate.

```
<?php echo mail_to('myaddress@mydomain.com', 'contact address', array('encode' => true)) ?>
// will generate in HTML something like
<a href="&#109;&#x61;&#105;&#108;&#x74;&#x6f;&#58;&#x6d;&#121;&#x61;&#x64;&#x64;&#114;&#101;&#x73
```

Image helper

Considering that all your HTML pages are handled by actions, the `href` attribute is not a worry anymore. But what about all those tags having a `src` attribute? Symfony has image and javascript helpers that are faster to write and do most of the job themselves.

In addition, future versions will support configuration for the default location of media files. This may prove helpful if you decided suddenly to host all of your project media files (css, js, pdf, jpg, gif, etc.) in a new server to increase performance.

To create an `<img>` tag:

```
<?php echo image_tag('test') ?>
```

```
// will generate in HTML
<img href="/images/test.png" alt="Test" />
```

By default, the image `test.png` will be looked for in the `/images/` directory of the web server (which is the `myproject/web/image/` directory of your file structure, unless you customized it).

If you need to change the default file type or the default location, you can pass a more defined URL:

```
<?php echo image_tag('test.gif') ?>
// will generate in HTML
<img href="/images/test.gif" alt="Test" />

<?php echo image_tag('/my_images/test.gif') ?>
// will generate in HTML
<img href="/my_images/test.gif" alt="Test" />
```

You can add a second argument to specify additional HTML attributes; this argument must be an array.

```
<?php echo image_tag('/my_images/test.gif', array('class' => 'shortImage')) ?>
// will generate in HTML
<img href="/my_images/test.gif" alt="Test" class="shortImage"/>
```

Don't bother to specify a `alt` attribute if your media file has an explicit name, since symfony will determine it for you.

To fix the size of an image, use the attribute `size`:

```
<?php echo image_tag('/my_images/test.gif', array('size' => '100x20')) ?>
// will generate in HTML
<img href="/my_images/test.gif" alt="Test" width="100" height="20"/>
```

To keep the code readable and let you write templates fast, the options argument can use an abbreviated string syntax:

```
<?php echo image_tag('/my_images/test.gif', 'size=100x20 alt=My new car class=shortImage onclick=
// is the same as
<?php echo image_tag('/my_images/test.gif', array('size' => '100x20', 'alt' => 'My new car', 'cla
// will generate in HTML
<img href="/my_images/test.gif" alt="My new car" width="100" height="20" class="shortImage" oncli
```

JavaScript helper

To link to JavaScript files, you can omit the path and the suffix:

```
<?php echo javascript_include_tag('myscript') ?>
// will generate in HTML
<script language="javascript" type="text/javascript" src="/js/myscript.js">
```

Once again, the default location can be overridden in a function call for quick needs, or redefined for the whole site in the configuration files for major changes.

If you read the whole documentation, you will see that symfony gives you three different ways to require an additional JavaScript file. You should use them with this order of preference:

- parameter definition in the `view.yml` file: Most of the time, the best solution
- `$this->getResponse()->addJavascript()` method call from the action: When the name of the files to include depend on calculations made in the action
- `javascript_include_tag()` helpers in the template: To include a JavaScript file at a given position inside a template and not in the head.

The last solution is rarely good; if you need it, check whether your code needs some refactoring (for instance write a JavaScript function call in the code and require the script inclusion in the head). However, you sometimes don't have the choice, for instance if you have to include traffic measure scripts at the end of your pages.

> **Note**: You may be looking for a `stylesheet_tag()` helper, but there is none. If you need to include a custom stylesheet, you should not do it in the template but rather in the view or in the action, as described in the view configuration chapter. This way, the declaration appears in the `<head>` instead of in the `<body>`.

Absolute paths

The url and asset helpers generate relative paths by default. You can easily add an `absolute=true` parameter to force the output to absolute paths, for instance for inclusions of links in an email, a RSS feed, or an API response.

```
<?php echo url_for('module/action', true) ?>
<?php echo link_to('my link', 'module/action', 'absolute=true') ?>
<?php echo image_tag('test', 'absolute=true') ?>
<?php echo javascript_include_tag('myscript', 'absolute=true') ?>
```

Templating in practice : Form helpers

Overview

Symfony provides form helpers to avoid repeating code in templates that contain forms, especially when the default value is linked to the application Model.

Introduction

In templates, HTML tags of form elements are very often mixed with PHP code. Form helpers in symfony aim to simplify this task and to avoid opening `<?php echo` tags in the middle of an element tag. They also duplicate the `name` attribute to define the `id` value.

In addition, if form controls are linked to objects in your data model, symfony can automate the hassle of defining a default value or listing the possible values.

Symfony also provides helpers for form validation. A detailed presentation of this feature can be found in the form validation chapter.

Main form tag

Some helpers are here to facilitate the writing of form tags, especially since they are often mixed with php code.

To create a form, you have to use the symfony `form_tag` function since it transforms the action given as a parameter into a routed URL:

```
<?php echo form_tag('test/save') ?>
// will generate in HTML
<form method="post" action="http://www.mysite.com/index.php/test/save">
```

This will route the form to the `save` action of the `test` module. You can learn more about the URL management in the routing chapter.

To change the default `method`, the default `enctype` or to specify other attributes, use the second argument:

```
// options argument with the associative array syntax
<?php echo form_tag('test/save', Array('multipart' => true, 'id' => 'myForm')) ?>
// will generate in HTML
<form method="post" enctype="multipart/form-data" id="myForm" action="http://www.mysite.com/index

// options argument with the string syntax
<?php echo form_tag('test/save', 'method=get class=simpleForm') ?>
// will generate in HTML
<form method="get" class="simpleForm" action="http://www.mysite.com/index.php/test/save">
```

As there is no need for a closing form helper, you should use the HTML `</form>` tag, even if it doesn't look good in your source code.

Standard form elements

Each element in a form will have by default an `id` attribute equal to its `name` attribute. You can specify custom attributes by adding a last argument, using the associative array syntax or the string syntax.

Here are the standard form element helpers:

- text field (input):

```
<?php echo input_tag('name', 'default value') ?>
// will generate in HTML
<input type="text" name="name" id="name" value="default value" />
```

- long text field (textarea):

```
<?php echo textarea_tag('name', 'default content', 'size=10x20') ?>
// will generate in HTML
<textarea name="name" id="name" rows="20" cols="10">default content</textarea>
```

- checkbox:

```
<?php echo checkbox_tag('married', '1', true) ?>
<?php echo checkbox_tag('driverslicence', 'B', false) ?>
// will generate in HTML
<input type="checkbox" name="married" id="married" value="1" checked="checked" />
<input type="checkbox" name="driverslicence" id="driverslicence" value="B" />
```

- radio button:

```
<?php echo radiobutton_tag('status', 'value1', true) ?>
<?php echo radiobutton_tag('status', 'value2', false) ?>
// will generate in HTML
<input type="radio" name="status" value="value1" checked="checked" />
<input type="radio" name="status" value="value2" />
```

  Note that the `id` attribute is not defined by default with the value of the `name` attribute for the radio button helper. That's because you need to have several radio button tags with the same name to obtain the automated 'deselecting the previous one when selecting another' feature, and that would imply having several HTML tags with the same `id` attribute in your page - which is strictly forbidden.

- value selection field (select):

```
<?php echo select_tag('name', options_for_select(Array('0' => 'Steve', '1' => 'Bob', '2' =
// will generate in HTML
<select name="name" id="name">
  <option value="0">Steve</option>
  <option value="1">Bob</option>
  <option value="2">Albert</option>
  <option value="3" selected="selected">Ian</option>
  <option value="4">Buck</option>
</select>
```

  When the value is the same as the text to display, use an array instead of an associative array

```
<?php echo select_tag('payment', options_for_select(Array('Visa', 'Eurocard', 'Mastercard'
```

```
// will generate in HTML
<select name="payment" id="payment">
  <option selected="selected">Visa</option>
  <option>Eurocard</option>
  <option>Mastercard</option>
</select>
```

Multiple selection field uses the same syntax, and the selected values can then be an array

```
<?php echo select_tag('payment', options_for_select(Array('Visa', 'Eurocard', 'Mastercard'
// will generate in HTML
<select name="payment" id="payment" multiple="multiple">
  <option selected="selected">Visa</option>
  <option>Eurocard</option>
  <option selected="selected">Mastercard</option>
</select>
```

When the data needed to set the value and the text to display can be obtained via methods of an object, you can simply pass an array of the objects, and the names of the methods to use to retrieve the value and display text.

```
<?php echo select_tag('articles', objects_for_select($articles, 'getId', 'getTitle', 1) )
```

This will iterate through the `$articles` array, using the current object by calling the `getId()` method for the value, and `getTitle()` for the display text.

- field to upload a file:

```
<?php echo input_file_tag('name') ?>
// will generate in HTML
<input type="file" name="name" id="name" value="" />
```

- field to enter a password:

```
<?php echo input_password_tag('name', 'value') ?>
// will generate in HTML
<input type="password" name="name" id="name" value="value" />
```

- hidden field:

```
<?php echo input_hidden_tag('name', 'value') ?>
// will generate in HTML
<input type="hidden" name="name" id="name" value="value" />
```

- submit button (as text):

```
<?php echo submit_tag('Save') ?>
// will generate in HTML
<input type="submit" name="submit" value="Save" />
```

- submit button (as image):

```
<?php echo submit_image_tag('submit_img') ?>
// will generate in HTML
<input type="image" name="submit" src="/images/submit_img.png" />
```

This last helper uses the same syntax and has the same advantages as the `image_tag` helper, described in the link helpers chapter.

Rich form elements

Dates

Forms are often used to get **dates**. Date format is also culture dependent, and wrong dates is the main reason for form validation failures. The `input_date_tag` can assists the user with a javascript calendar:

```
<?php echo input_date_tag('dateofbirth', '2005-05-03', 'rich=true') ?>
// will generate in HTML
// a text input tag together with a calendar widget
```

The last argument specifies that the rich form has to be used. If it was set to `false` (which is the default value), the normal date entry widget (three text inputs for day, month and year) would be used in place.

The accepted date values are the ones recognized by the `strtotime()` php function. Beware that, in some limit cases, the results can be surprising:

```
// Can be used

// work fine
<?php echo input_date_tag('test', '2006-04-01', 'rich=true') ?>
<?php echo input_date_tag('test', 1143884373, 'rich=true') ?>
<?php echo input_date_tag('test', 'now', 'rich=true') ?>
<?php echo input_date_tag('test', '23 October 2005', 'rich=true') ?>
<?php echo input_date_tag('test', 'next tuesday', 'rich=true') ?>
<?php echo input_date_tag('test', '1 week 2 days 4 hours 2 seconds', 'rich=true') ?>
// returns NULL
<?php echo input_date_tag('test', null, 'rich=true') ?>
<?php echo input_date_tag('test', '', 'rich=true') ?>

// Not to be used

// date zero = 01/01/1970
<?php echo input_date_tag('test', 0, 'rich=true') ?>
// non-English date formats don't work
<?php echo input_date_tag('test', '01/04/2006', 'rich=true') ?>
```

Rich text

Rich text editing in a textarea is also possible, since symfony provides an interface with the TinyMCE widget that inputs html code.

In order to use TinyMCE, you have to download it first from this page and unpack it in a temporary folder. Copy the `tinymce/jscripts/tiny_mce` directory into your project `web/js/` directory, and add to your `settings.yml`:

```
all:
  .settings:
    rich_text_js_dir:  js/tiny_mce
```

Now you can run the rich version of the textarea input tag:

```
<?php echo textarea_tag('name', 'default content', 'rich=true size=10x20')) ?>
// will generate in HTML
// a rich text edit zone powered by TinyMCE
```

If you want to specify custom options for TinyMCE, use the `tinymce_options` option:

```
<?php echo textarea_tag('name', 'default content', 'rich=true size=10x20 tinymce_options=language
```

Country

You will probably need to display a **country selection field**. The `select_country_tag` helper can do it for you; it uses the default culture to choose in which language the countries will be displayed.

```
<?php echo select_country_tag('country', 'Albania') ?>
// will generate in HTML
<select name="country" id="country">
  <option>Afghanistan</option>
  <option selected="selected">Albania</option>
  <option>Algeria</option>
  <option>American Samoa</option>
  <option>Andorra</option>
  ...
```

Form helpers for objects

In many cases form elements are linked to fields in a database table. As you use an object-relational mapping, the form elements are actually linked to attribute accessors and setters. Symfony provides an alternate version to all the previous helpers, where the `$value` argument - which defines the default value - is replaced by two arguments, `$object` and `$method`. The method must be the attribute accessor of the object. The method name is also given to the `name` and `id` attributes of the generated HTML tag, so the first argument is suppressed.

The simple text input tag now looks like:

```
<?php echo object_input_tag($customer, 'getTelephone') ?>
// will generate in HTML
<input type="text" name="getTelephone" id="getTelephone" value="0123456789" />
```

...provided that `$customer->getTelephone() = '0123456789'`.

So simply add the `object_` prefix in front of the name of the form helpers to get the name of the related object form helper:

```
object_input_tag($object, $method, $options)
object_textarea_tag($object, $method, $options)
object_checkbox_tag($object, $method, $options)
...
```

> **Note**: Contrary to the regular for helpers, the object form helpers are only available if you declare explicitly the use of the `Object` helper in your template:

```
<?php echo use_helper('Object') ?>
```

The most useful helper of this type is the `object_select_tag`, since it can get the list of values from another table. For instance, imagine that you have an `Article` and an `Author` table, where one or more articles are written by one author. This means that the `Article` table has a `author_id` column:

Article	Author
id	id
author_id	name
title	

Now, picture a form built to edit the data of an article. There is an easy way to get the list of the author names when editing the `AuthorId` column:

```
object_select_tag($article, 'getAuthorId', 'related_class=Author')
```

Symfony will use the `->toString()` method of the `Author` class to display the names of the authors in a list. If the method is undefined, the primary key (in the example, the `id` column) of the `Author` table is used instead.

By default, the list of authors will be the complete list ordered by creation date, but you can specify another `AuthorPeer` method to display a custom array of authors, in the order you want, with the `peer_method` parameter:

```
object_select_tag($article, 'getAuthorId', 'related_class=Author, peer_method=getFamousAuthorsOrd
```

A few other options can define the content of an object select tag. `include_title` and `include_blank` allow to add a title or a first empty line to the list:

```
object_select_tag($article, 'getAuthorId', 'related_class=Author')
object_select_tag($article, 'getAuthorId', 'related_class=Author include_blank=true')
object_select_tag($article, 'getAuthorId', 'related_class=Author include_title=true')
//will display three selects with lists of values looking like:
-------      -------      -------
|Steve |     |      |     |Author|
|John  |     |Steve |     |Steve |
|Mark  |     |John  |     |John  |
-------      |Mark  |     |Mark  |
             -------      -------
```

> **Note**: the `object_select_tag` helper uses the Propel layer; it might not work if you use another data abstraction layer.

Easy object update

The object relational mapping given by Propel allows you to easily modify a record in a table from a form if the fields of the forms are properly named.

For instance, if you have an object `Author` with attributes `name`, `age` and `address`, and that you created a template looking like:

```
<?php echo form_tag('author/update') ?>
Name : <?php echo object_input_tag($author, 'getName') ?><br />
```

```
Age : <?php echo object_input_tag($author, 'getAge') ?><br />
Address:<br />
<?php echo object_textarea_tag($author, 'getAddress') ?>
</form>
```

Then the `update` action of the `author` module can simply use the `fromArray` modifier:

```
public function executeUpdate ()
{
  $author = AuthorPeer::retrieveByPk($this->getRequestParameter('id'));

  $this->forward404Unless($author instanceof Author);

  $author->fromArray($this->getRequest()->getParameterHolder()->getAll(), Author::TYPE_FIELDNAME)
  $author->save();

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

JavaScript helpers

Overview

Interactions on the client side, complex visual effects or asynchronous communication are common in web 2.0 applications. All that require JavaScript, but coding it by hand is often cumbersome and long to debug. Fortunately, symfony automates many of the common uses of JavaScript in the templates.

Introduction

JavaScript has long been considered as a candy for beginners, without any real use in professional web applications due to the lack of compatibility between browsers. In the late 90s, the word 'DHTML' meant 'unreadable HTML code with embedded JavaScript written three times for compatibility', and it sometimes referred to dynamic web pages that didn't work.

These days are now over, since the compatibility issues are (mostly) solved and some robust libraries allow to program complex interactions in JavaScript without the need for 200 lines of code and 20 hours of debugging. The most popular advance is called AJAX, and will be treated a little further.

For now, all you need to know is... JavaScript. Or maybe forget a little about it, because what will be shown below is how NOT to use JavaScript to do what you would expect JavaScript for. Just reconsider this scripting language as an opportunity to manipulate the DOM on the client side, with all the power that it gives.

All the helpers described here are available in templates provided you declare the use of the `Javascript` helper:

```
<?php echo use_helper('Javascript') ?>
```

Some of them output HTML code, some of the output JavaScript code. It will be specified for each of them.

Basic helpers

Clean XHTML JavaScript inclusion

If you want to write a piece of JavaScript code in your page, instead of writing manually something like:

```
<script type="text/javascript">
//<![CDATA[
  function foobar()
  {
  ...
  }
//]]>
</script>
```

You could try the `javascript_tag()` function:

```
<?php echo javascript_tag("
```

```
  function foobar()
  {
  ...
  }
") ?>
```

Link behavior

The most common use of JavaScript is in a hyperlink that triggers a particular script. In HTML, you would probably write it like that:

```
<a href="#" onClick="alert('foobar');return none;">Click me!</a>
```

Symfony proposes a helper to write it in a more concise way, the `link_to_function()`:

```
<?php echo link_to_function("Click me!", "alert('foobar')") ?>
```

Like the `link_to()` URL helper, you can add options to the `<a>` tag generated by a `link_to_function()`:

```
<?php echo link_to_function("Click me!", "alert('foobar')", "class=mylink style=color:#f00 alt=pl
```

> **Note**: Just like the `link_to()` helper has a `button_to()` brother, you can trigger a JavaScript from a button (`<input type="button">`) by calling the `button_to_function()` helper. And if you prefer a clickable image, just call `link_to_function(image_tag('myimage'), "alert('foobar')")`.

Updating an element

One common task in applications that use JavaScript a lot is the update of an element in the page. This is something that you usually write:

```
<?php echo javascript_tag("
  document.getElementById("indicator").innerHTML = "<strong>Evacuation complete</strong>";
") ?>
```

Symfony provides a helper that produces JavaScript - not HTML - for this purpose, and it's called `update_element_function()`:

```
<?php echo javascript_tag(
  update_element_function('indicator', array(
    'content'  => "<strong>Evacuation complete</strong>",
  ))
) ?>
```

You may ask: What's the interest of a helper if it is longer to write than the normal code? Sometimes, you want to insert content *before* an element, or *after* it; you may also want to *remove* an element instead of just updating it, or to do nothing according to a condition. In these cases, the JavaScript code becomes somehow messier, but the `update_element_function()` keeps very readable:

```
// insert content just after the 'indicator' element
```

```
update_element_function('indicator', array(
  'position' => 'after',
  'content'  => "<strong>Evacuation complete</strong>",
));

// remove the element before the 'indicator' element, and only if $condition is true
update_element_function('indicator', array(
  'action'   => $condition ? 'remove' : 'empty',
  'position' => 'before',
))
```

The helper makes your templates easier to understand than any JavaScript code, and you have one single syntax for similar behaviours. That's also why the helper name is so long: It makes the code self-sufficient, without the need of extra commentaries.

Graceful degradation

You already know how to qualify some HTML code to make it executed only by browsers with no JavaScript support, uting the `<noscript>` tags. What if you could do it the other way around, i.e. qualify some code so that it is executed only by browsers actually supporting JavaScript? Symfony provides two helpers just for that, and it facilitates the creation of applications that degrade gracefully.

```
<?php if_javascript(); ?>
  <p>You have JavaScript enabled.</p>
<?php end_if_javascript(); ?>

<noscript>
  <p>You don't have JavaScript enabled.</p>
</noscript>
```

> **Note**: You don't need to include the `echo` when calling these two helpers.

AJAX helpers

What if the function updating an element of the page was not in a JavaScript, but a PHP script executed by the server? This would give you the opportunity to change part of the page according to a server response. The `remote_function()` does exactly that:

```
<?php echo javascript_tag(
  remote_function(array(
    'update'  => 'myzone',
    'url'     => 'mymodule/myaction',
  ))
) ?>
```

When called, this script will update the element of id `myzone` with the response or the request of the `mymodule/myaction` action. This kind of interaction is called AJAX, and it's the heart of highly interactive web applications. Here is how Wikipedia describes them:

> *AJAX makes web pages feel more responsive by exchanging small amounts of data with the server behind the scenes, so that the entire web page does not have to be reloaded each time*

> *the user makes a change. This is meant to increase the Web page's interactivity, speed, and usability.*

AJAX relies on XMLHttpRequest, a JavaScript object that behaves like a hidden frame, which you can update from a server request and reuse to manipulate the rest of your web page.

An AJAX interaction is made up of three parts: a caller (a link, a button or any control that the user manipulates to launch the action), a server action, and a zone in the page to display the result of the action to the user. Symfony provides multiple helpers to insert AJAX interaction in your templates by putting the caller in a link, a button, a form, or a clock. Beware that this time, these helpers output HTML code, not JavaScript.

For instance, imagine a link that would call a remote function. Symfony writes it like this:

```
<?php echo link_to_remote('Delete this post', array(
    'update' => 'indicator',
    'url'    => 'post/delete?id='.$post->getId(),
)) ?>

<?php echo link_to_remote(image_tag('refresh'), array(
    'update' => 'emails',
    'url'    => '@list_emails',
)) ?>
```

Notice that the `url` parameter can contain either an internal URI (`module/action?parameters`) or a routing rule, just like in a regular `link_to()`. The askeet tutorial has other examples of `link_to_remote()` uses.

> **Note**: Actions called as remote functions know that they are in an AJAX transaction, and therefore automatically don't include the web debug toolbar in development and skip the decoration process (i.e. their template is **not** included in a layout by default). This means that if you want an AJAX view to be decorated, you need to specify explicitely `layout: foobar` for this view in the module `view.yml`.

The second popular way of interacting with web page is forms. Forms normally call a remote function, but this causes the refreshing of the whole page. The correspondence of the `link_to_function()` for a form would be that the form submission only updates an element in the page with the response of the server. This is what the `form_remote_tag()` helper does.

```
<?php echo form_remote_tag(array(
    'update'   => 'item_list',
    'url'      => '@item_add',
)) ?>
  <label for="item">Item:</label>
  <?php echo input_tag('item') ?>
  <?php echo submit_tag('Add') ?>
</form>
```

A `form_remote_tag()` opens a `<form>`, just like the `form_tag()` helper does. You are invited to use the symfony form helpers in AJAX forms as well as in regular forms.

If you want to allow a form to work both in page mode and in AJAX mode, the best solution is to define it like a regular form, but to provide, in addition to the normal submit button, a second button (`<input`

type="button" />) to submit the form in AJAX - symfony calls it a submit_to_remote(). This will help you build AJAX interactions that degrade gracefully.

```
<?php echo form_tag('@item_add_regular') ?>
  <label for="item">Item:</label>
  <?php echo input_tag('item') ?>
  <?php echo submit_tag('Add') ?>
  <?php echo submit_to_remote('ajax_submit', 'Add in AJAX', array(
      'update'   => 'item_list',
      'url'      => '@item_add',
  )) ?>
</form>
```

Modern forms can also react not only when submitted, but also when the value of a field is being updated by a user. In symfony, you use the observe_field() helper for that:

```
<?php echo form_tag('@item_add_regular') ?>
  <label for="item">Item:</label>
  <?php echo input_tag('item') ?>
  <?php echo submit_tag('Add') ?>
  <?php echo observe_field('item', array(
      'update'   => 'item_suggestion',
      'url'      => '@item_being_typed',
  )) ?>
</form>
```

The module/action written in the @item_being_typed rule will be called each time the item field changes, and the action will be able to get the current item value from the value request parameter. If you want to pass something else than the value of the observed field, you can specify it as a JavaScript expression in the with parameter. For instance, if you want the action to get a param parameter, write the observe_field() helper as follows:

```
<?php echo observe_field('item', array(
      'update'   => 'item_suggestion',
      'url'      => '@item_being_typed',
      'with'     => "'param=' + value",
  )) ?>
```

Note that this helper doesn't output an HTML element, but a behaviour for the element passed as parameter. You will see below more examples of JavaScript helpers assigning behaviours.

If you want to observe all the fields of a form, you'd better use the observe_form() helper, which calls a remote function each time one of the form fields is modified.

Last but not least, the periodically_call_remote() helper is an AJAX interaction triggered every x seconds. It is not attached to a HTML control, but runs transparently in the background, as a behaviour of the whole page. This can be of great use to track the position of the mouse, autosave the content of a large textarea, etc.

```
<?php echo periodically_call_remote(array(
    'frequency' => 60,
    'update'    => 'notification',
    'url'       => '@watch',
    'with'      => "'param=' + $('mycontent').value",
```

```
)) ?>
```

If you don't specify the number of seconds (`frequency`) to wait between two calls to the remote function, the default value of 10 is taken.

> **Note**: The AJAX helpers won't work if the URL of the remote action doesn't belong to the same domain as the current page. This restriction exists for security reasons, and relies on browsers limitations that cannot be bypassed.

Prototype

Prototype is a great JavaScript library that extends the possibilities of the client scripting language, adds the missing functions you've always dreamt of, and offers new mechanisms to manipulate the DOM.

The Prototype files are present in the symfony framework `data/web/sf/js/` directory. This means that you can use Prototype by adding in your action:

```
$this->getResponse()->addJavascript('/sf/js/prototype/prototype');
```

...or by adding it in the `view.yml`:

```
all:
  javascripts: [/sf/js/prototype/prototype]
```

> **Note**: Since the symfony AJAX helpers themselves use Prototype, as soon as you add a remote call to a template through a symfony helper, you also include automatically the Prototype library.

Once the Prototype library is loaded, you can take advantage of all the new functions it adds to the JavaScript core. It is not the purpose of this documentation to describe them all, and you will easily find on the web good documentations of Prototype (for instance ParticleTree's doc, Sergio Pereira's doc or the one from script.aculo.us). But as our latest example used the dollar (`$()`) function, let's just have a look at it:

```
// In JavaScript,
node = $("elementID");
// means the same as
node = document.getElementById("elementID");
```

Other than being short and sweet, the `$()` function makes it short to find something in the DOM, and it is also more powerful than `document.getElementById()` because of its ability to retrieve multiple elements.

```
allNodes = $("firstDiv", "secondDiv");
```

Because programming in JavaScript with Prototype is much more fun than doing it by hand, and because it is also part of symfony, you should really spend a few minutes to read the related documentation.

Remote calls parameters

All the AJAX helpers described above can take other parameters, in addition to the `update` and `url` parameters.

Position

Just like you did in the `update_element_function()` helper, you can specify the element to update as relative to a specific element by adding a `position` parameter:

```
<?php echo link_to_remote('Delete this post', array(
    'update'   => 'indicator',
    'url'      => 'post/delete?id='.$post->getId(),
    'position' => 'after',
)) ?>
```

This will insert the result of the AJAX call after the `indicator` element. With this method, you can do several AJAX calls and see the answers pile down the `update` element.

The `position` parameter can be defined as:

Value	Position
`before`	Before the element
`after`	After the element
`top`	at the top of the content of element
`bottom`	at the bottom of the content of element

Conditions

A remote call can take an additional parameter to allow a confirmation of the user before actually submitting the XMLHttpRequest:

```
<?php echo link_to_remote('Delete this post', array(
    'update'   => 'indicator',
    'url'      => 'post/delete?id='.$post->getId(),
    'confirm'  => 'Are you sure?',
)) ?>
```

A JavaScript dialog box showing 'Are you sure?' will pop-up when the user clicks on the link, and the `post/delete` action will be called only if the user confirms his choice by clicking 'Ok'.

The remote call can also be conditioned by a test performed on the browser side (in JavaScript):

```
<?php echo link_to_remote('Delete this post', array(
    'update'    => 'indicator',
    'url'       => 'post/delete?id='.$post->getId(),
    'condition' => "$('elementID') == true",
)) ?>
```

Method

By default, AJAX requests are made in `post` mode. If you want to make an AJAX call that doesn't modify data, or if you want to display a form that has built-in validation as the result of an AJAX call, you might need to change the AJAX request method to `get`. This is simpy done through the `method` option:

```
<?php echo link_to_remote('Delete this post', array(
    'update'    => 'indicator',
    'url'       => 'post/delete?id='.$post->getId(),
    'method'    => 'get',
)) ?>
```

Script execution

If the response code of the AJAX call (the code sent by the server, inserted in the `update` element) contains JavaScripts, you might be surprised to see that these scripts are not executed by default. This is to prevent remote attack risks, except if the developper knows for sure what code is in the response.

That's why you have to declare the ability to execute scripts in remote responses explicitly with the `script` option:

```
// If the response of the post/delete action contains JavaScript, allow them to be executed by th
<?php echo link_to_remote('Delete this post', array(
    'update' => 'indicator',
    'url'    => 'post/delete?id='.$post->getId(),
    'script' => true,
)) ?>
```

If the remote response contains AJAX helpers (like, for instance, `remote_function()`), be aware that these PHP functions generate JavaScript code, and won't execute unless you add the `'script' => true` option.

> **Note**: Even if you enable script execution for the remote response, you won't actually see the scripts in the remote code - that is, if you use a tool to check the generated code. The scripts will execute but will not appear in the code. Although peculiar, this behaviour is perfectly normal.

Callbacks

One important drawback of AJAX interactions is that they are invisible for the user until the zone to update is actually updated. This means that in case of slow network, or server failure, the user may believe that his action was taken into account while it was not.

This is why it is important to notify the users of the events of an AJAX interaction.

By default, each remote request is an asynchronous process during which various JavaScript callbacks can be triggered (for progress indicators and the likes). All callbacks get access to the 'request' object, which holds the underlying XMLHttpRequest. The callbacks correspond to the events of any AJAX interaction:

Callback	Event
`before`	before request is initiated
`after`	immediately after request was initiated and before `loading`
`loading`	when the remote document is being loaded with data by the browser
`loaded`	when the browser has finished loading the remote document
`interactive`	when the user can interact with the remote document, even though it has not finished loading
`success`	when the XMLHttpRequest is completed, and the HTTP status code is in the 2XX range
`failure`	when the XMLHttpRequest is completed, and the HTTP status code is not in the 2XX range
`404`	when the url returns a 404 status
`complete`	when the XMLHttpRequest is complete (fires after success/failure if they are present)

For instance, it is very common to show a loading indicator when a remote call is initiated, and to hide it once the response is received. In a `link_to_remote()`, you can write it like that:

```
<?php echo link_to_remote('Delete this post', array(
    'update'   => 'indicator',
    'url'      => 'post/delete?id='.$post->getId(),
    'loading'  => "Element.show('indicator')",
    'complete' => "Element.hide('indicator')",
)) ?>
```

The `show` and `hide` methods, as well as the JavaScript `Element` object, are another useful addition of Prototype.

Visual effects

Symfony integrates the visual effects of the script.aculo.us library, to allow you to do more than show and hide divs in your web pages. You will find a good documentation on the effects syntax in the script.aculo.us wiki. Basically, they are JavaScript objects and functions that manipulate the DOM in order to achieve complex visual effects:

```
// highlights the element 'my_field'
Effect.Highlight('my_field', {startcolor:'#ff99ff', endcolor:'#999999'})

// Blinds down an element
Effect.BlindDown('id_of_element');

// Fades away an element
Effect.Fade('id_of_element',
  { transition: Effect.Transitions.wobble })
```

Have a look at the combination effects demo to get a visual idea of what these functions do.

Symfony encapsulates these JavaScript objects in a helper called `visual_effects()`. It outputs JavaScript that can be used in a regular link:

```
<?php echo link_to_function("Show the secret div", visual_effect('appear', 'secret_div')) ?>
// will make a call to Effect.Appear('secret_div')
```

The `visual_effects()` helper can also be used in the AJAX callbacks:

```
<?php echo link_to_remote('Delete this post', array(
    'update'   => 'indicator',
    'url'      => 'post/delete?id='.$post->getId(),
    'loading'  => visual_effect('appear', 'indicator'),
    'complete' => visual_effect('fade', 'indicator').visual_effect('highlight', 'post_nb'),
)) ?>
```

Notice how you can combine visual effects by concatenating them in a callback.

JSON

AJAX helpers allow to update more than one element at a time, using the JSON syntax. For more information about it, check out the user submitted documentation in the symfony wiki.

Complex interactions

JavaScript also provides tools to build up complex interactions. But the complexity of the implementation makes it quite rare over the Internet, although examples like google suggest (for autocomplete input) or netvibes (for sortible lists) show that desktop-like interactions are now possible in web applications.

The script.aculo.us library provides high-level JavaScript functions that symfony integrates as helpers. They allow the following kind of interaction:

- **Autocomplete input**: A text input that shows a list of words matching the user's entry while he/she types in. Examples of the use of the `input_auto_complete_tag()` helper in symfony can be found in the askeet tutorial.
- **Drag and Drop**: The ability to grab an element, move it and release it somewhere else, for a real "put it there" feeling. Refer to the shopping cart tutorial for a step-by-step documentation on how to use `draggable_element()` and `drop_receiving_element()` - and a self explaining screencast.
- **Sortable lists**: Another possibility offered by draggable elements is the ability to sort a list by moving its items. The `sortable_element()` helper adds the 'sortable' behaviour to an item, and the symfony book has a dedicated sortable chapter dealing with this feature.
- **Edit in place**: More and more web applications allow users to edit the content of pages directly in the page, without the need to redisplay the content in a form (see a demo here. the symfony helper that does it is the `input_in_place_editor_tag()`.

Templating in practice : Internationalization helpers

Overview

Displaying localized content is not a worry in Symfony thanks to the internationalization helpers. They will transparently process localized content (dates, numbers, currencies) according to the user culture.

Date helpers

Symfony simplifies the display of dates and numbers according to the user culture. For an international website, all dates and numbers should be output through a internationalization filter, and that's what the following helpers do.

All these helpers have a culture argument, which is the last argument and is optional. If omitted, the helper uses the user culture instead. To learn more about cultures and internationalization, read the i18n chapter.

First, declare the need of the helper at the top of your template:

```
<?php use_helper('Date') ?>
```

Here are two functions aimed at date and date plus time display:

```
<?php echo format_date($date) ?>
<?php echo format_date($date, 'd', 'en') ?>

<?php echo format_datetime($date) ?>
```

The second argument is the format, that can be specified either with predefined patterns:

-	-	-
d	ShortDatePattern	MM/dd/yyyy
D	LongDatePattern	dddd, dd MMMM yyyy
F	FullDateTimePattern	dddd, dd MMMM yyyy HH:mm:ss
m, M	MonthDayPattern	MMMM dd
r, R	RFC1123Pattern	ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
s	SortableDateTimePattern	yyyy'-'MM'-'dd'T'HH':'mm':'ss
t	ShortTimePattern	HH:mm
T	LongTimePattern	HH:mm:ss
Y	YearMonthPattern	yyyy MMMM

Or directly with the basic keys.

For instance, to display the month with 2 digits and the year with 4, you can write:

```
<?php echo format_date($date, 'MM/yyyy') ?>
```

And if you have a date interval, another helper manages all the cases for you:

```
<?php echo format_daterange($start_date, $end_date, 'MM/yy', 'from %s to %s', 'starting from %s',
```

The last three arguments specify the text to be used when:

- two dates exist
- only the start date is given
- only the end date is given

The `%s` are replaced by the formatted dates.

For instance, if `$start_date` is the 1st of April 2005 and `$end_date` the 3rd of July 2005, the displayed text will be `from 04/05 to 07/05`.

Number helpers

It is also possible to format a number:

```
<?php use_helper('Number') ?>

<?php echo format_number(12000.10) ?>
// will generate in HTML for an American user
12,000.10
// will generate in HTML for a French user
12 000,10
```

To format a monetary amount, use the `currency_format` function. It takes as optional argument the ISO currency code of your amount.

```
<?php echo format_currency(12000.10, 'EUR') ?>
// will generate in HTML for an American user
â ¬12,000.10
// will generate in HTML for a French user
12 000,10 â ¬
```

The last two helpers are here to return a text version of the interval separating two dates:

```
<?php echo distance_of_time_in_words($from_time, $to_time, $include_seconds = false) ?>
<?php echo time_ago_in_words($from_time, $include_seconds = false) ?>
```

Country and language names

You will probably need to display country names, for instance in addresses. Country names differ according to the culture (for instance, 'Great Britain' in French is 'Grande-Bretagne'). If the data format is a two-digit ISO code (like 'GB' for Great-Britain), the country and language helpers do the job for you.

In order to use the internationalization helpers, you must include the `I18N` helper module.

```
<?php use_helper('I18N') ?>
```

```
<?php echo format_country('GB') ?>
// will generate in HTML for an American user
Great Britain
// will generate in HTML for a French user
Grande-Bretagne

<?php echo format_language('EN') ?>
// will generate in HTML for an American user
English
// will generate in HTML for a French user
anglais
```

Countries and languages are often displayed in selection lists for data input in forms. If you do so, you shouldn't need the previous helpers since Sensio provides a `select_country_tag` helper, described in the form helpers chapter.

Templating in practice : Other helpers

Overview

Symfony offers a few helpers to quickly manipulate text.

Text helpers

truncate_text()

For no-wrap width-limited blocks - for instance in `select` tags - you may need to allow a maximum number of characters and truncate the rest if needed. To that purpose, you should use the `truncate_text` helper:

```
<?php use_helper('Text') ?>

<?php echo truncate_text($text, $length , $truncate_string = '...') ?>

// For instance, if $text="You can't stop me ! You can't stop me ! You can't stop me ! You can't
<?php echo truncate_text($text, 30) ?>
// will be output as
You can't stop me ! You can't ...
```

excerpt_text()

Websites that offer full-text search need to show where the text was found. This is another type of truncation, and the helper that does it is `excerpt_text`:

```
<?php echo excerpt_text($text, $sentence, $radius = 100 , $truncate_string = '...') ?>

// For instance, if $text="Finding a word in an ocean of text can sometimes be a real pain."
<?php echo excerpt_text($text, 'ocean', 10) ?>
// will be output as
...ord in an ocean of text c...
```

highlight_text()

Maybe you need to highlight a sentence in a text:

```
<?php echo highlight_text($text, $sentence, [$highlighter]) ?>
//$highlighter is the class 'highlight' by default

// For instance, if $text='The best PHP framework is symfony'
<?php echo highlight_text($text, 'symfony') ?>
// will be output as
The best PHP framework is <span class="highlight">symfony</span>
```

simple_format_text()

Or maybe you need to format an ASCII text to HTML:

```
<?php echo simple_format_text($text) ?>
```

`simple_format_text` surrounds paragraphs with `<p>...</p>` tags, and converts line breaks into `<br />`. Two consecutive newlines (\n\n) are considered as a paragraph, one newline (\n) is considered a line break, three or more consecutive newlines are turned into two newlines. It might be useful to display, for instance, data entered by the user in a textarea.

auto_link_text()

Another useful helper finds all the URLs in a text and turns them into hyperlinks

```
<?php echo auto_link_text('my homesite is www.mysite.com') ?>
// will generate in HTML
my homesite is <a href="http://www.mysite.com">www.mysite.com</a>
```

Parameter Holders

Overview

Many of the objects available in symfony have the ability to be extended. They host a parameter holder, in which new data can be kept. This can be particularly useful to quickly add a custom attribute without overriding the full class.

Introduction

When dealing with web requests, user sessions or custom business objects like shopping carts, the need for extension often arises. If the extension carries a logic, it has to be done by extending (or overriding) the class, but if the need is simply data storage, the solution is the parameter holder.

The `sfRequest`, `sfUser` and `sfShoppingCart` objects, among others, have an attribute of class `sfParameterHolder` in which additional data can be stored and retrieved without any need to override the class.

> **Note**: the addition of an attribute holder to an object is made by a composition of the `sfParameterHolder` and the object, rather than by inheritance. This avoids problems during further extensions of the classes.

The parameter holder class

The `sfParameterHolder` object is an extensible repository for data of any kind (string, associative array, object, etc.).

The `->set($key,$value)` and `->get($key, $default_value)` methods are used to set a new parameter and get its value.

```
$myPH = new sfParameterHolder();

$myPH->set('weather','sunny');
$myPH->set('temperature','cool');

$myPH->get('weather','no idea')   =>  sunny
$myPH->get('temperature')         =>  cool
```

The `->has($key)` method is used to check if a parameter exists.

```
$myPH->has('temperature')         =>  true
$myPH->has('humidity')            =>  false
```

The `->getAll()` method is used to get all the parameters as an associative array. The `->getNames()` method is used to get only the names of the parameters hold:

```
$myPH->getAll()                   =>   { 'weather' => 'sunny' , 'temperature' => 'cool' }
$myPH->getNames()                 =>   [ 'weather' , 'temperature' ]
```

If you need to add several parameters at once, use the `->add()` method with an associative array as argument:

```
$myParameters = { 'humidity' => 0.8, 'forecast' => $forecast };
$myPH->add($myParameters);
```

Use the `->remove($key)` method to delete one parameter and its key, or the `->clear()` method to reset the whole parameter holder.

```
$myPH->remove('temperature');
$myPH->has('temperature')          =>  false
```

Namespaces

Parameter holders also have the ability to deal with namespaces. The namespaces are completely isolated from each other. When no namespace is defined, as in the examples above, the default namespace 'symfony/default' is used.

```
$myPH->set('weather','sunny');
$myPH->set('weather','foggy','symfony/my/namespace');

$myPH->get('weather')                                 =>  sunny
$myPH->get('weather',null,'symfony/my/namespace')  =>  foggy
```

> **Note**: when you do a `->get()` over a namespace, the second argument (the default value) is required.

The `->getAll()` method is limited to a namespace. Without any argument, it returns the parameters of the default namespace. If a namespace is passed as an argument, only the parameters of this namespace are returned:

```
$myPH->getAll('symfony/my/namespace') =>  { 'weather', 'foggy' }
```

The `->hasNamespace($ns)`, `->getNamespaces()` and `->removeNamespace($ns)` work as you imagine they do:

```
$myPH->hasNamespace('symfony/my/namespace')       =>  true
$myPH->hasNamespace('symfony/other')              =>  false

$myPH->getNamespaces()                            => ['symfony/default','symfony/my/namespace']

$myPH->removeNamespace('symfony/my/namespace');
$myPH->has('weather',null,'symfony/my/namespace') =>  false
```

If you want that all the `sfParameterHolder` methods use a default namespace different from 'symfony/default', you have to set it during initialization:

```
$mySpecialPH = new sfParameterHolder('symfony/special');
$mySpecialPH->getNamespaces()                     => ['symfony/special']
```

sfRequest parameter holders

The `sfRequest` object has two prebuilt parameter holders, and convenient shortcuts to their content:

- the **ParameterHolder** contains the parameters of the request (passed as GET, POST or via the symfony routing policy). They can be read but not written.

```
$myPH = $this->getRequest()->getParameterHolder();
```

- the **AttributeHolder** can pass data between the action and the template, or between several actions; it is erased at every new request

```
$myPH = $this->getRequest()->getAttributeHolder();
```

They are both of class `sfParameterHolder`.

As a matter of fact, the logic of the parameter holders is almost hidden since the `sfAction` and the `sfRequest` objects have shortcuts that bypass the parameter holder logic.

```
$param = $this->getRequestParameter('param','default');
//is equivalent to
$param = $this->getRequest()->getParameter('param','default');
//is equivalent to
$param = $this->getRequest()->getParameterHolder()->get('param','default');

$this->getRequest()->setAttribute('attrib', $value);
//is equivalent to
$this->getRequest()->getAttributeHolder()->set('attrib', $value);

$attrib = $this->getRequest()->getAttribute('attrib');
//is equivalent to
$attrib = $this->getRequest()->getAttributeHolder()->get('attrib');
```

For instance, if you need to get the parameter `id` in a request called by:

```
http://myapp.example.com/index.php/module/article/action/read/id/234
// or
http://myapp.example.com/index.php/module/article/action/read?id=234
// or
http://myapp.example.com/index.php/module/article/action/read
//with id=234 in the POST HTTP header
```

You simply need to write, in the action `read` of the `article` module:

```
$id = $this->getRequestParameter('id',null);
```

Now let's imagine that you need to display a list in two columns. The switch to the second column must occur after the fifth element is displayed. Your template would look like:

```
<table><tr><td><ul>
<?php $sf_request->setAttribute('itemsdisplayed', 0) ?>
<?php foreach ($items as $item): ?>
  <li>Item <?php echo $item->to_s() ?></li>
  <?php $sf_request->setAttribute('itemsdisplayed', $sf_request->getAttribute('itemsdisplayed')+1
```

```
  <?php if ($sf_request->getAttribute('itemsdisplayed')>5): ?>
     </ul></td><td><ul>
     <?php $sf_request->setAttribute('itemsdisplayed', -50) ?>
  <?php endif ?>
<?php endforeach ?>
</ul></td></tr></table>
```

sfUser parameter holders

The `sfUser` object also has two parameter holders:

- the **ParameterHolder** is not persistent; it is erased at every new request

  ```
  $myPH = $this->getUser()->getParameterHolder();
  ```
- the **AttributeHolder** keeps data throughout requests; you should use it as soon as you need persistent session data

  ```
  $myPH = $this->getUser()->getAttributeHolder();
  ```

As for the `sfRequest` object, the `sfUser` objects has shortcut methods to access its parameter holders:

```
[php]
 $this->getUser()->setAttribute('attrib', $value);
 //is equivalent to
 $this->getUser()->getAttributeHolder()->set('attrib', $value);
```

Other objects

Lots of other objects in the symfony framework also have one parameter holder. For instance:

- `sfShoppingCart`
- `sfShoppingCartItem`
- `sfPropelPager`
- `sfFilter`
- ...

For these objects, the common `->getAttribute()` and `->setAttribute()` shortcut methods are available.

How to configure a web server

Overview

There is more than one way to configure a web server to enable it to access symfony applications. This chapter illustrates the different configuration possibilities and some tricks to optimize your access.

Introduction

In the examples described above, the `myproject` project contains a `myapp` application. The front controller of this application is called `index.php` and lies in `/home/steve/myproject/web/`. The symfony data directory is `$data_dir`.

Virtual host

Virtual hosting allows you to setup your web server so that your symfony application appears at the root of a domain (or a sub domain):

```
http://myapp.example.com/
```

Let's assume that you run an Apache server. To setup a virtual host for the `myapp` application, add the following lines to the `httpd.conf` file:

```
<Directory "/$data_dir/symfony/web/sf">
 AllowOverride All
 Allow from All
</Directory>
<VirtualHost *:80>
  ServerName myapp.example.com
  DocumentRoot "/home/steve/myproject/web"
  DirectoryIndex index.php
  Alias /sf /$data_dir/symfony/web/sf

  <Directory "/home/steve/myproject/web">
   AllowOverride All
   Allow from All
  </Directory>
</VirtualHost>
```

The `Alias` statement in the middle is necessary for the images of the debug sidebar to be displayed. The `$data_dir` placeholders has to be replaced by your PEAR data directory. For example, for *nix, you should type:

```
Alias /sf /usr/local/lib/php/data/symfony/web/sf
```

You will find more about the PEAR directories in the installation chapter.

Restart Apache, and that's it: the webapp can now be called and viewed through a standard web browser at the URL:

```
http://myapp.example.com/
```

or, in debug mode:

```
http://myapp.example.com/myapp_dev.php/
```

URL rewriting

By default, the web server is configured to avoid mentioning the production front controller (`index.php`) in the URL. This means that instead of displaying:

```
http://myapp.example.com/index.php/
```

your server displays and recognizes:

```
http://myapp.example.com/
```

This uses the `mod_rewrite` Apache module, and requires the following lines to be present in the `myproject/web/.htaccess` (which is the case by default):

```
# all files with .something are skipped
RewriteCond %{REQUEST_URI} \..+$
RewriteCond %{REQUEST_URI} !\.html$
RewriteRule .* - [L]
# the others are redirected to the front web controller
RewriteRule ^(.*)$ index.php [QSA,L]
```

There is one more cosmetic addition that you may wish to add to your URLs : a `.html` at the end. Normally, when calling the `index` action of the `main` module, with default routing configuration, the URL displayed would be:

```
http://myapp.example.com/main/index
```

The default `settings.yml` of the application contains a commented `suffix` parameter:

```
all:
#  .settings:
#    suffix:        .
```

In order to make the `index` action of the `main` module appear like:

```
http://myapp.example.com/main/index.html
```

You can uncomment and set the `suffix` parameter to `.html`:

```
all:
  .settings:
    suffix:        .html
```

Alias

If you already have a website on a domain name, and if you wish that your symfony application can be accessed within this domain, then the virtual host solution cannot work. For instance, let's assume that you want to access our symfony application with:

```
http://www.example.com/myapp/
```

To do that, open the `httpd.conf` and add the following lines:

```
Alias /myapp/ /home/steve/myproject/web/
<Directory "/home/steve/myproject/web">
   AllowOverride All
   Allow from All
</Directory>
```

You also need to edit the `.htaccess` file located in your `myproject/web/` directory and change the last rewrite rule from

```
RewriteRule ^(.*)$ index.php [QSA,L]
```

to

```
RewriteRule ^(.*)$ /myapp/index.php [QSA,L]
```

Restart Apache, and try to access your webapp:

```
http://www.example.com/myapp/
```

Multiple applications within one project

During the course of project design, you will meet one day or another the problem of multiple applications access. For instance, in our `myproject` project, you might have a `myapp` application for the public and an `admin` application o manage the content (usually called a back-office). How to authorize access to multiple applications within one project ?

Virtual hosts

You can add a new virtual host in your Apache `httpd.conf` That's fairly easy to understand:

```
<Directory "/$data_dir/symfony/web/sf">
 AllowOverride All
 Allow from All
</Directory>

<VirtualHost *:80>
  ServerName myapp.example.com
  DocumentRoot "/home/steve/myproject/web"
  DirectoryIndex index.php
  Alias /sf /$data_dir/symfony/web/sf
```

```
  <Directory "/home/steve/myproject/web">
   AllowOverride All
   Allow from All
  </Directory>
</VirtualHost>

<VirtualHost *:80>
  ServerName admin.example.com
  DocumentRoot "/home/steve/myproject/web"
  DirectoryIndex admin.php
  Alias /sf /$data_dir/symfony/web/sf

  <Directory "/home/steve/myproject/web">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

Aliases

Alternatively, you can add a new alias. This will allow you to have separated web contents (.css, .js, images, etc.) for each application. This also avoids direct modification of the `httpd.conf` file.

First, create a new directory in your `web` directory:

```
$ mkdir /home/steve/myproject/web/admin
```

Then, move the front controllers of the `admin` application to this new directory, and copy the `.htaccess` to have one for this new app:

```
$ cd /home/steve/myproject/web
$ mv admin.php admin/index.php
$ mv admin_dev.php admin/
$ cp .htaccess admin/
```

Then, execute the two last steps steps described above to setup the alias. Edit the `.htaccess` file located in your `myproject/web/admin/` directory and change

```
RewriteRule ^(.*)$ index.php [QSA,L]
```

to

```
RewriteRule ^(.*)$ /admin/index.php [QSA,L]
```

Eventually, edit both front controllers (`myproject/web/admin/index.php` and `myproject/web/admin/admin_dev.php`) and change:

```
define('SF_ROOT_DIR', realpath(dirname(__FILE__).'/..'));
```

to:

```
define('SF_ROOT_DIR', realpath(dirname(__FILE__).'/../..'));
```

And this is enough to be able to access the `admin` application by:

```
http://whateveryourmainurlis/admin/
```

> **Note**: you will need to recreate the same file structure in your `web/admin` directory as in a classic `web` directory (with `css`, `js`, `images`, `uploads` directories) since all the paths to the root now point to this `admin/` directory.

IIS

Symfony is compatible with IIS. To learn all about the installation and configuration of a symfony project in a IIS environment, read the related tutorial.

Installing symfony on IIS

Yes, there's something else than Apache for web servers. That's why we'll see in this tutorial how to install symfony on IIS.

What do we need?

- A windows server (really)
- An IIS server (obviously)
- PHP5 installed and configured in IIS as an isapi module (see PHP5 installation instructions)
- isapi/rewrite/ www.isapirewrite.com, an URL manipulation engine we'll use to replace Apache's mod_rewrite. Why this product? It seems to be mainly used on IIS, and the free lite version works for symfony. If you're used to something else, read on, there should be only one difference in the last chapter of this tutorial.

Install symfony

First, update the pear package, since you need the version 1.4.0 to handle channels:

```
$ pear upgrade PEAR
```

Then, add the symfony channel:

```
$ pear channel-discover pear.symfony-project.com
```

Install symfony beta version >= 0.5.73:

```
$ pear install symfony/symfony-beta
```

> **Note**: if you don't have the `phing` package, you will need to install it as well:
>
> ```
> $ pear install http://phing.info/pear/phing-current.tgz
> ```

Find more about symfony installation.

Initialize the project

To create the directory of your project (let's use `c:\myproject`) and the basic tree structure for an application called `myapp`, open a command console and type :

```
$ cd c:\myproject
$ symfony init-project myproject
$ symfony init-app myapp
```

Configure IIS

We'll consider two configurations options from now on:

- In the first case, the webserver is only used for our symfony project, and the URL is something like `http://myproject/`. Assuming the directory in which you created the directory `myapp` is `c:\myproject\`, configure the root directory of your server to be `c:\myproject\web` (in IIS administration console).
- The other option is to install our symfony project in a directory (may be virtual) of you server, the URL is in this case something like `http://myserver/myproject/`. In IIS administration console, create a new virtual directory on the root of your website for the directory `c:\myproject\web`.

Add a virtual directory in the main directory of your server. Name it `sf` and configure it to `data\symfony\web\sf` of your pear directory. If you installed php5 in `c:\php5` with default configuration, the full path is `C:\php5\PEAR\pear\data\symfony\web\sf`.

Configuration of URL rewriting

We'll assume that isapi/rewrite is installed and running on your server. We have not yet bought it, so we only have one httpd.ini file to configure. The configuration depends on the options defined previously:

- for URLs like `http://myproject/` we need:

```
# Defend your computer from some worm attacks
RewriteRule .*(?:global.asa|default\.ida|root\.exe|\.\.).* . [F,I,O]


# we skip all files with .something except .html
RewriteCond URL .*\..+$
RewriteCond URL (?!.*\.html$).*
RewriteRule (.*) $1 [L]


# we keep the .php files unchanged
RewriteRule (.*\.php)(.*) $1$2 [L]


# finally we redirect to our front web controller
RewriteRule (.*) /index.php [L]
```

- for URLs like `http://myserver/myproject/` we need:

```
# Defend your computer from some worm attacks
RewriteRule .*(?:global.asa|default\.ida|root\.exe|\.\.).* . [F,I,O]


# we skip all files with .something except .html
RewriteCond URL /myproject/.*\..+$
RewriteCond URL (?!/myproject/.*\.html$).*
RewriteRule /myproject/(.*) /myproject/$1 [L]
```

```
# we keep the .php files unchanged
RewriteRule /myproject/(.*\.php)(.*) /myproject/$1$2 [L]


# finally we redirect to our front web controller
RewriteRule /myproject/(.*) /myproject/index.php [L]
```

You may restart IIS.

Configuring symfony

The last step is editing the file `settings.yml` located in the config directory of our symfony project (`c:\myproject\apps\myapp\config\` in our example). You won't be surprised to find a little difference between our two options, so:

- for URLs like `http://myproject/`, we need these lines:

```
all:
 .settings:
   path_info_key:          HTTP_X_REWRITE_URL
```

- for URLs like `http://myserver/myproject`, we need these lines:

```
all:
 .settings:
   path_info_key:          HTTP_X_REWRITE_URL
```

Important note: if you don't use isapi/rewrite/, the `HTTP_X_REWRITE_URL` may be wrong. You'll have to make a specific test to know how to configure symfony. Open the file `myapp_dev.php` (or `whatever you named your application_dev.php`) in the web directory of your project, and add these following lines on line 2 :

```
print phpinfo();
die();
```

Now open the following URL : `http://myproject/myapp_dev.php/test/rewrite` (or `http://myserver/myproject/myapp_dev.php/test/rewrite` depending your configuration), and look at the PHP Variables. If `_SERVER["PATH_INFO"]` equals /test/rewrite, remove the `path_info_key` line from `settings.yml`, else you'll have to find the name of the variable containing this value (`HTTP_X_REWRITE_URL` for `isapi/rewrite/`). Remove the previous two lines, symfony is ready.

Configuring a symfony application in its own directory

A symfony project may contain several applications. In the default configuration, all the applications share the same css and uploads directory. Let's change this and create a specific directory for our application myapp. Considering our previous options, we'll manage to have these URLs : `http://myproject/myapp` and `http://myserver/myapp`. This time, there are no differences in the configuration between these options.

Create a subdirectory `myapp` in the web directory of your symfony project (to have in our case `c:\myproject\web\myapp`).

Copy the files `index.php` and `myapp_dev.php` in this directory, create there two directories named `css` and `uploads`.

In IIS administration console, create a new virtual directory on the root of your server named `myapp`. Configure this virtual directory to `c:\myproject\web\myapp`.

Add these lines at the end of your `httpd.ini` file:

```
#[Configuration for direct myapp access]
 # we skip all files with .something except .html
 RewriteCond URL /myapp/.*\..+$
 RewriteCond URL (?!/myapp/.*\.html$).*
 RewriteRule /myapp/(.*) /myapp/$1 [L]

 # we keep the .php files unchanged
 RewriteRule /myapp/(.*\.php)(.*) /myapp/$1$2 [L]

 # finally we redirect to our front web controller
 RewriteRule /myapp/(.*) /myapp/index.php [L]
```

Edit the file `settings.yml` of `myapp` application (`c:\myproject\apps\myapp\config\settings.yml`) and make sure you have the following lines:

```
all:
  .settings:
    path_info_key:          HTTP_X_REWRITE_URL
```

(see previous note concerning `HTTP_X_REWRITE_URL`)

Finally, edit both front controllers (`index.php` and `myapp_dev.php` in `c:\myproject\web\myapp`) and change:

```
define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/..'));
```

to:

```
define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/../..'));
```

That's it, you're done.

How to setup a routing policy

Overview

One of the main problems faced by frameworks is the aspect of the generated URLs : they are generally long, complex and not search engine friendly. SymFony introduces a new *routing* system that brings you total control on the urls of your applications.

Introduction

Let's consider the case of a blogging application where users publish articles. In symfony, in order to display an article, you would need a URL looking like:

```
http://myapp.example.com/index.php/article/read/id/100
```

This URL calls the `read` action of the `article` module with an `id` parameter taking he value 100.

In order to optimize the way the search engines index the pages of dynamic websites, and to make the URLs more readable, some blogging tools propose a *permalink* feature. A permalink is simply a defined and permanent URL address aimed at browser bookmarks and search engines. In the previous example, the permalink could look like:

```
http://myapp.example.com/index.php/article/permalink/title/my_article_title
```

The only difference with the first URL is the use of more descriptive keywords. The `permalink` action will have to transform the `title` argument into an article `id`, by looking in a table of permalinked pages, to point correctly to the previous action.

This process could be pushed further to display URLs even more simple and explicit, for instance like:

```
http://myapp.example.com/article/my_article_title
// or why not
http://myapp.example.com/2005/06/25/my_article_title
```

The most common way to address this need is to use the `mod_rewrite` module of the Apache server, together with URL rewriting rules. These rules transform the URLs into something that Apache can understand before submission of the request:

1. Apache receives a request for `http://myapp.example.com/2005/06/25/my_article_title`
2. `mod_rewrite` transforms this URL into `http://myapp.example.com/index.php/article/read/title/my_article_title`
3. Now Apache knows that it has to execute `index.php` with `/article/read/title/my_article_title` as a value for the `path_info` argument

However, this method has two serious issues:

- you need an Apache server and the `mod_rewrite` module

- the rewriting is only one-way

As a matter of fact, if you wish to create an URL to this article, you will need to transform manually the base URL into a "smart" URL. The input URLs (handled by `mod_rewrite`) and the output URLs (handled by the application) are completely unrelated.

Symfony can natively transform output URLs and interpret input URLs. Consequently, you can create bijective associations between URLs and the Front Controller. This rewriting, called *routing* in symfony, relies on a configuration file called `routing.yml` that can be found in the `config/` directory of every application.

Routing input URLs

Rules and patterns

The `routing.yml` contains **rules**, or bijective associations between a URL pattern and the "real" request parameter. A typical rule file contains:

- a label, which is there for legibility and can be used by the link helpers
- an `url` key showing the pattern to be matched
- a `param` used to set default values for some of the arguments of the "real" call

Here is an extract of the `routing.yml` file that illustrates the rewriting of our example blog URL:

```
article_by_title_with_date:
  url:    /:year/:month/:day/:title
  param:  { module: article, action: permalink }
```

The rule stipulates that every request showing the pattern `/:year/:month/:day/:title` will have to be transformed into a call to the `permalink` action of the `article` module with arguments `year`, `month`, `day` and `title` taken from the base URL.

So the example URL

```
http://myapp.example.com/index.php/2005/06/25/my_article_title
```

Will be understood as if written

```
http://myapp.example.com/index.php/article/permalink/year/2005/month/06/day/25/title/my_article_t
```

...and call the `permalink` action of the `article` module with the following arguments:

```
$this->getRequestParameter('year') => 2005
$this->getRequestParameter('month') => 06
$this->getRequestParameter('day') => 25
$this->getRequestParameter('title') => my_article_title
```

Let's add a second rule to handle URLs like:

```
http://myapp.example.com/index.php/article/100
```

Simply add the following lines in the `routing.yml` file:

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read }
```

Notice that in the pattern, the word `article` is a string whereas `id` is a variable (because it starts with a `:`).

> **Note**: you smart readers may have guessed that as soon as a rule such as the one mentionned above is added, the default rule (which is `/:module/:action/*`) will not work anymore with the `article` module, because the module name will match the pattern `/article/:id` first. If you start creating rules with strings that match the names of your modules, you probably need to change the default rule to something like:
>
> ```
> default:
>     url:   /action/:module/:action/*
> ```

Pattern constraints

Now what if you needed to have access to articles from their title:

```
http://myapp.example.com/index.php/article/my_article_title
```

Well, this looks problematic. This URL should be routed to the `permalink` action, but it already satisfies the `articl_by_id` rule and will be automatically routed to the `read` action. To solve this issue, each entry can take a third parameter called `requirements` to specify constraints in the pattern (in the shape of a regular expression). That means that you can modify the previous rule to route URLs to `read` only if the `id` argument is an integer:

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read }
  requirements: { id: ^\d+$ }
```

Now you can add a third rule to gain access to articles from their title:

```
article_by_title:
  url:          /article/:permalink
  param:        { module: article, action: permalink }
```

Rules are ordered and the routing engine takes the first one that satisfies the pattern and the pattern constraints. That's why you don't need to add a constraint to the last rule (specifying that `permalink` can not be an integer):

- `/article/100` matches the first rule and will be handed to `read`
- `/article/my_article_title` doesn't match the first rule but matches the second, so it will be handed to `permalink`

Now that you know about pattern constraints, that would be a good thing to add some to the very first rule:

```
article_by_title_with_date:
```

```
  url:          /:year/:month/:day/:title
  param:        { module: article, action: permalink }
  requirements: { year: ^\d{4}$, month: ^\d\d$, day: ^\d\d$ }
```

The routing engine allows you to handle a large set of rules; however, you have to add the most precise constraints and order them properly so that no ambiguity may arise.

> **Hint**: the YAML syntax allows you to write more legible configuration files if you write associative arrays line by line. For instance, the last rule can also be written:

```
article_by_title_with_date:
  url:        /:year/:month/:day/:title
  param:
    module:   article
    action:   permalink
  requirements:
    year:     ^\d{4}$
    month:    ^\d\d$
    day:      ^\d\d$
```

Default values

Here is a new example:

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read, id: 1 }
```

This rule defines the default value for the `id` argument. This means that a `/article/100` URL will behave as previously, but in addition, the URL `/article/` will be equivalent to `/article/1`. The default parameters don't need to be variables found in the pattern. Consider the following example:

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read, id: 1, display: true }
```

The `display` argument will be passed with the value `true`, whatever the pattern. And, if you look carefully, you will see that `article` and `read` are also default values for variables not found in the pattern.

Default rules

The default `routing.yml` has a few default rules. To allow the old style 'module/action' URLs to work:

```
default:
  url:   /:module/:action/*
```

As mentionned above, you may need to change this rule if some of your modules have names that can match other patterns.

The other default rules are used to set the root URL to point the default module and action:

```
homepage:
```

```
  url:   /
  param: { module: #SF_DEFAULT_MODULE#, action: #SF_DEFAULT_ACTION# }

default_index:
  url:   /:module
  param: { action: #SF_DEFAULT_ACTION# }
```

The default module and action themselves are configured in the `settings.yml` file.

How to avoid mentionning the front controller ?

In all previous examples, the URLs still have contain the `index.php` header to be processed. This is because the front controller has to be called first so that the routing feature can work.

If you have the `mod_rewrite` module activated, use the following configuration (which is the default configuration bundled with symfony in the `myproject/web/.htaccess` file) to tell apache to call the `index.php` file by default:

```
Options +FollowSymLinks +ExecCGI

RewriteEngine On

# we skip all files with .something
RewriteCond %{REQUEST_URI} \..+$
RewriteRule .* - [L]

# we check if the .html version is here (caching)
RewriteRule ^$ index.html [QSA]
RewriteRule ^([^.]+)$ $1.html [QSA]
RewriteCond %{REQUEST_FILENAME} !-f

# if no rule matched the url, we redirect to our front web controller
RewriteRule ^(.*)$ index.php [QSA,L]

# big crash from our front web controller
ErrorDocument 500 "<h2>Application error</h2>SymFony application failed to start properly"
```

Now a call to

```
http://myapp.example.com/article/read/id/100
```

Will be properly understood as

```
http://myapp.example.com/index.php/article/read/id/100
```

Outputting smart URLs

Matching patterns

Until now, the `routing.yml` file only helped to reproduce the `mod_rewrite` behaviour, i.e. understanding properly formatted URLs. The good news is, now that you defined routing rules, they will be automatically used to transform URLs *from* your application.

In symfony, when you write a link in a template, you use the `link_to()` helper:

```
<?php echo link_to($article->getTitle(), '/article/read?id='.$article->getId()) ?>
```

To read more about this helper, check the chapter about link helpers. With the default routing configuration, this outputs the following HTML code:

```
<a href="/index.php/article/read/id/100">my_article_title</a>
```

But since you wrote routing rules, symfony will automatially interpret them in the other way and generate:

```
<a href="/index.php/article/100">my_article_title</a>
```

The rules will be parsed with the same order as for the interpretation of an input request, and the first rule matching the arguments of the `link_to()` second argument will determine the pattern to be used to create the output URL.

Getting rid of `index.php`

As for now, the link helpers still output the name of the front controller. If the web server is configured to handle calls without mention of the front controller, as described above, the routing system can be told not to include it.

This is done in the application `settings.yml` configuration file. To turn off the display of the front controller in the production environment, write:

```
prod:
  .settings
    no_script_name:  on
```

Adding a `.html`

Having an output URL like:

```
http://myapp.example.com/2005/06/25/my_article_title
```

is not bad, but

```
http://myapp.example.com/2005/06/25/my_article_title.html
```

is much better. It changes the way your application is perceived by the user, from "a dynamic thing with cryptic calls" to "a deep and well organized web directory". All that with a simple suffix. In addition, the search engines will grant more stability to a page named like that.

As before, this is simply done in the `settings.yml` configuration file of the application:

```
prod:
  .settings
    suffix:          .html
```

The default suffix is set to `.`, which means that nothing is appended to the end of the routed url. You can specify any type of suffix, including `/` to have an URL looking like:

```
http://myapp.example.com/2005/06/25/my_article_title/
```

It is sometimes necessary to specify a suffix for a unique routing rule. In that case, directly write the suffix in the related `url:` line of the `routing.yml` file; the global suffix will be ignored.

```
article_list_feed:
  url:          /latest_articles.rss
  param:        { module: article, action: list, type: rss }

update_directory:
  url:          /updates/
  param:        { module: update, action: list }
```

Retrieve information about the current route

If you need to retrieve information about the current route, for instance to prepare a future 'back to page xxx' link, you should use the methods of the `sfRouting` object. For instance, if your `routing.yml` defines:

```
my_rule:
  url:   /call_my_rule
  param: { module: mymodule, action: myaction }
```

Use the following calls in the action:

```
// if you require an URL like
http://myapp.example.com/call_my_rule/param1/xxx/param2/yyy

$uri = sfRouting::getInstance()->getCurrentInternalUri();
// will return 'mymodule/myaction?param1=xxx&param2=yyy'

$uri = sfRouting::getInstance()->getCurrentInternalUri(true);
// will return '@myrule?param1=xxx&param2=yyy'

$route = sfRouting::getInstance()->getCurrentRouteName();
// will return 'myrule'
```

The URIs returned by the `->getCurrentInternalUri()` method can be used in a call to a `link_to()` helper.

In addition, you might want to get the first or the last action called in a template. The following variables are automatically updated at each request and are available to templates:

```
$sf_first_action
$sf_first_module
$sf_last_action
$sf_last_module
```

You might ask: Why can't I simply retrieve the current module/action ? Because the calls to actions is a stack, and several calls can be made for a unique request. For instance, if the end of an action contents a `forward` statement, several actions will be called.

How to speed up a site with the caching system

Overview

The powerful and flexible symfony cache system can speed up a website by saving chunks of generated HTML code, or even full pages, for future requests. Easy to set up thanks to YAML files, the cache can also be cleared with a simple command.

Introduction

The principle of HTML caching is simple: part or all of the HTML code that is sent to a user upon a request can be reused for a similar request, so this HTML code is stored in a special place (the `cache` folder in symfony), where the front controller will look for it before executing an action. If a cached version is found, it is sent without executing the action, thus greatly speeding up the process. If there is no cached version found, the action is executed, and its result (the view) is stored in the cache folder for future requests.

As all the pages may contain dynamic information, the HTML cache is disabled by default. It is up to the site administrator to activate it in order to improve performance.

Symfony handles four different types of HTML cache:

- cache of an action
- cache of a partial, a component, or of a component slot
- cache of an entire page
- cache of a fragment of a template

The three first types are handled with YAML configuration files, the last one is managed by calls to helper functions in the template.

The symfony cache system uses files stored on the web server hard disk. This keeps the cache simple and efficient, without any other prerequisites than the framework itself. It is not yet possible to cache in memory or in a database.

Global cache settings

For each application of a project, the HTML cache mechanism can be activated or deactivated completely per environment. In the `settings.yml` configuration file, notice the `cache` parameter:

```
prod:

dev:
  .settings:
    ...
    cache:                  off
    ...
```

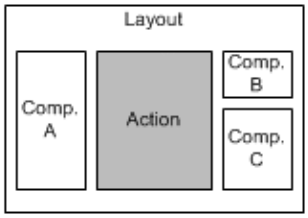
The default value of this parameter is set to `off`, so you have to specifically set it to `on` to enable it.

It is set to `off` by default in the development environment. This means that if you decide to add caching to one of your apps, you will not be able to see the effect of it in the development environment with the default configuration.

Consequently, the boost given by HTML caching is only perceptible in the production environment - or in any other environment where `cache: on`.

One other cache parameter can be changed in the `settings.yml` file: The default cache lifetime i.e., the number of seconds after which a cached file is overwritten and the page processed again. The default `default_cache_lifetime` is set to one day, or 86400 seconds.

Caching an action



Slot cache

Actions that display static information (i.e. without any call to a database) or actions that read the information in a database but without modifying it (typically GET requests) are often good clients for caching.

For instance, imagine an action that returns the list of all the users of your website (`user/list`). Unless a user is modified, added or removed (and this matter will be discussed later), this list always displays the same information, so it is a good candidate for caching.

The result of the above-mentioned action is a processed template (`listSuccess.php`), and this is what is going to be cached.

To activate the cache on such actions, simply add a `cache.yml` file in the `myproject/apps/myapp/modules/user/config/` directory, with the following content:

```
list:
  activate:   on
  type:       slot

all:
  lifetime:   86400
```

This configuration stipulates that the cache is `on` for the `list` action, and that it is of the `slot` type (i.e. caching an action, as opposed to caching the whole page, which will be described later). The `lifetime` is the time (in seconds) after which the page will be processed again and the cached version replaced.

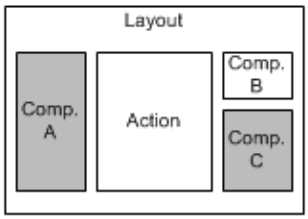
Now, if you try to call this action from your browser (probably by requesting an URL like `http://myapp.example.com/user/list`), you will notice no difference the first time, but refreshing the page will probably show a notable boost in response time.

The caching system also works for pages with arguments. The `user` module may have, for instance, a `show` action that expects an `id` argument to display the details of a user. You would then just need to add the following lines at the top of the module `cache.yml` file:

```
show:
  activate:    on
  type:        slot
```

Now, requests like `http://myapp.example.com/user/show/id/12` will create new records in the cache folder and if you repeat this request, it will be much faster the second time.

Caching a partial, a component, or a component slot



Component cache

The view chapter explains how to reuse code fragments across several templates, using the `include_partial()` helper.

```
<?php include_partial('mymodule/mypartial') ?>
```

A partial is as easy to cache as an action. To activate it, create a `cache.yml` in the partial module `config/` directory (in the example above, in `modules/mymodule/config/`) and activate the cache for the partials by declaring their names with a leading underscore:

```
_mypartial:
  activate: on

all:
  activate: off
```

Now all the templates using this partial won't actually execute the PHP code of the partial, but use the cached version instead.

> **Note**: The `slot` type cache is more powerful than the partial cache, since when an action is cached, the template is not even executed - and if the template contains calls to partials, theses calls are not performed. Therefore, the partial caching is useful only if you don't use action caching in the calling action.

Just like for actions, partial caching is also relevant when the result of the partial depends on parameters:

```
<?php include_partial('mymodule/my_other_partial', array('foo' => 'bar')) ?>
```

A component is a lightweight action put on top of a partial. A component slot is a component for which the action varies according to the calling actions. These two inclusion types are very similar to partials, and support caching the same way. For instance, if your global layout shows a:

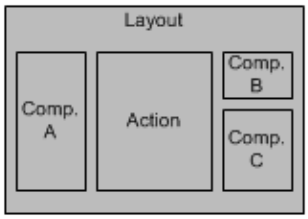
```
<?php include_component('global/day') ?>
```

...in order to show the current date, then you can cache this component with such a `cache.yml`:

```
_day:
  activate: on
```

> **Note**: Global components (the ones located in the application `templates/` directory) can be cached, provided you declare the activation in the application `cache.yml` file.

Caching an entire page



Page cache

The final page is the combination of the template and the layout, according to the decorator design pattern. Until now, all that was written in the cache was the template, so every request ended up in a decoration process (putting the cached template into a processed layout). But if the layout has no dynamic element, you can cache the whole page instead of just the template.

Let's say this is the case for the test application described above: the layout simply contains navigation links, nothing dynamic, and can easily be cached. The cache can then be set to `page` type instead of `slot`. Modify the `cache.yml` as follows:

```
show:
  activate:   on
  type:       page

list:
  activate:   on
  type:       page

all:
  lifetime:   86400
```

If you request again the aforementioned pages:

```
http://myapp.example.com/user/list
http://myapp.example.com/user/show/id/12
```

...the full pages will be completely cached, and the second time you request them, the response will be even faster than with the action result cached.

But even a cached page involves some PHP code execution. For such a page, symfony still loads the configuration, builds the response, etc. If you are really sure that a page is not going to change for a while, you can bypass symfony completely by putting the resulting HTML code directly into the `web/` folder. This works thanks to the Apache `mod_rewrite` settings, provided that your routing rule specifies a pattern ending with no suffix or with `.html`.

For instance, for a page accessible at:

```
http://myapp.example.com/user/list
```

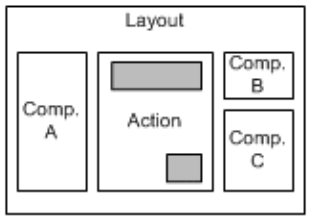
Create a `user/` directory in your project `web/` directory. Add in a `list.html` file containing the generated HTML code of the page, that you can get, for instance, by calling [curl](http://curl.haxx.se/.

```
$ cd web
$ mkdir user
$ cd user
$ curl http://myapp.example.com/user/list.html > list.html
```

Now, everytime that the `user/list` action is requested, Apache finds the corresponding `web/user/list.html` page and bypasses symfony completely. The trade-off is that you can't pilot the page cache with symfony anymore (lifetime, automatic deletion), but the speed gain is very important.

Unfortunately, the layout often contains some dynamic elements, including component slots. So the cases where the whole page can be cached are not very common. As a matter of fact, this type is often most used for RSS feeds, pop-ups, or pages with a layout that don't depend on cookies.

Caching a template fragment



Fragment cache

Many times the `slot` and `page` types will be too large for the templates of an application. For instance, the list of users can show a link of the last accessed user, and this information is dynamic. Would that mean that nothing can be cached?

Thankfully not. Symfony allows you to cache fragments of a template with the `cache()` helper. The `listSuccess.php` template could be written as follows:

```
<!-- uncached HTML -->
<?php echo link_to('last accessed user', 'user/show?id='.$last_accessed_user_id) ?>
<!-- /uncached HTML -->
```

```
<?php if (!cache('users')): ?>
  <!-- cached HTML -->
  <?php foreach ($users as $user): ?>
    <?php echo $user->getName() ?>
    ...
  <?php end foreach ?>
  <!-- /cached HTML -->
  <?php cache_save() ?>
<?php endif ?>

<!-- uncached HTML -->
...
<!-- /uncached HTML -->...
```

Here's how it works: if a cached version of the fragment named 'users' is found, it is used to replace the code between the `<?php if (!cache('unique_fragment_name')): ?>` and the `<?php endif ?>` lines. If not, then the code between theses lines is processed and saved in the cache, identified with the unique fragment name ('users' in the example). The code not included between such lines is always processed and not cached.

Note that the `list` action must not have a `slot`- or `page`- type cache enabled, since this would bypass the whole template execution and ignore the fragment cache declaration. So either remove the `list` part of the `cache.yml` file, or write instead:

```
list:
  activate:   off
```

After requesting twice the `http://myapp.example.com/user/list` URL, notice that the second execution is faster than the first. However, the answer is not as fast as when the cache was set to `slot` or `page`, since the template is partially processed and the decoration is made for every request.

You can declare additional fragments in the same template; however, you need to give each of them a unique name so that the symfony cache system can find them afterwards.

Like the `slot` and `page` caching, cached fragments can be given a lifetime in seconds:

```
<?php if (!cache('users', 43200)): ?>
...
```

The default cache lifetime (86400 seconds/one day) is used if no parameter is given to the `cache()` function.

Cache file structure

You don't need to know how the cache files are structured to make it work, so jump to the next part if you are not a *what's-behind-the-curtain* kind of person.

You probably noticed that each symfony project has a `cache` folder. The tree structure of its subdirectories is:

```
cache/[APP_NAME]/[ENV_NAME]/
```

... in which you can find three directories, `config`, `i18n` and `template`. As mentioned in the introduction, the HTML cache is stored in the `template` directory, so for the `myapp` application in the `prod` environment, focus on the `myproject/cache/myapp/prod/template/` directory.

It contains a tree structure that matches the requested URLs. In the examples above, you requested:

```
http://myapp.example.com/user/list
http://myapp.example.com/user/show/id/12
```

So the tree structure of `myproject/cache/myapp/prod/template/` is:

```
myapp.example.com/
  user/
    list/
    show/
      id/
        12/
```

That's as simple as it can be. The domain name is part of this path because you can host one app in different domains, and you don't want the cache of one domain to mess up with the others (imagine a multilingual website with different domain names like `myapp.example.com` and `myapp.example.fr`: they need distinct cache folders).

The files stored in these folders depend on the type of caching used:

cache type	file name
slot, partial, components	slot.cache
page	page.cache
fragment	fragment_users.cache
	fragment_other_unique_name.cache
	...

The name given to a fragment in the `cache()` helper is concatenated after `fragment_`. Here, the second fragment was generated by:

```
<?php if (!cache('other_unique_name', SF_DEFAULT_CACHE_LIFETIME)): ?>
...
```

If you use component or component slots in the layout, remember that their cache files are found in a tree structure relative to the module/action of the slot, not the module/action initially called.

Feel free to browse the cache folder to look at the chunks of code that are saved; you might feel more comfortable with the cache if you see that it only saves what it is supposed to save.

Removing something from the cache

Clearing the cache for all actions

When you modify a template or an action, you will need to clean the cache to avoid errors. The symfony command line can launch a `clear-cache` process which will erase all the cache (`config` and `i18n` included, see below). From the root folder of a project, call:

```
$ symfony clear-cache
// or use the short syntax
$ symfony cc
```

In the production environment, once the cache is cleared, the first request will reprocess the configuration files and recreate the `config` cache. Then, if the HTML cache is set to on for some actions, the first call to these actions will regenerate the corresponding cache.

You may want to be more specific about what to clear:

```
$ symfony clear-cache myapp
// will erase only the cache of the myapp application
$ symfony clear-cache myapp template
// will erase only the HTML cache of the myapp application
$ symfony clear-cache myapp config
// will erase only the config cache of the myapp application
```

This is pretty straightforward, but maybe a little too drastic for some cases.

Clearing the cache for specific actions

When the model is updated, the cache of the actions related to this model have to be cleared. Imagine that the `update` action of the `user` module modifies the columns of the `User` object. The caching of the `list` and `show` action has to be cleared, or else the old version of the templates, with erroneous data, will be displayed. This is where the `->remove()` method of the `sfViewCacheManager` object (a singleton) is useful.

In the `update` action of the `user` module, you need to add:

```
public function executeUpdate()
{
  ...
  $this->getContext()->getViewCacheManager()->remove('user/list');
  $this->getContext()->getViewCacheManager()->remove('user/show?id='.$this->getRequestParameter('
  ...
}
```

The `->remove()` method expects the same kind of target as you would provide a `url_for()` helper, so it is quite easy to figure out how to clear the cached files. As an optional second argument, it accepts a type precision to target specific files:

```
// removes all files
$this->getContext()->getViewCacheManager()->remove('user/list');
// removes only the page.cache file
$this->getContext()->getViewCacheManager()->remove('user/list', 'page');
// removes only the slot.cache file
$this->getContext()->getViewCacheManager()->remove('user/list', 'slot');
```

```
// removes only the fragment_users.cache files
$this->getContext()->getViewCacheManager()->remove('user/list', 'fragment_users');
```

The trickiest part is to determine which actions are influenced by a change in an object of the model. For instance, imagine that the current application has a `publication` module where publications are listed (`list` action) and described (`show` action), along with the details of their author (an instance of the `User` class). Modifying one User record will affect all the publication descriptions of which this user is the author. This means that you need to add to the `update` action of the `user` module something like:

```
$c = new Criteria();
$c->add(PublicationPeer::AUTHOR_ID, $this->getRequestParameter('id'));
$publications = PublicationPeer::doSelect($c);

foreach ($publications as $publication)
{
  $this->getContext()->getViewCacheManager()->remove('publication/show?id='.$publication->getId('
}
$this->getContext()->getViewCacheManager()->remove('publication/list');
```

When you start using the HTML cache, you need to keep a clear view of the dependencies of the actions, so that new errors don't appear because of a misunderstood relationship. Keep in mind that all the actions that modify the model should probably contain a bunch of calls to the `->remove()` method if the HTML cache is used somewhere in the application.

Clearing the cache of another application

If you deal with several applications and do caching in at least one of them, you will probably end up with the problem of clearing the cache of another application.

Just picture a back-office application that modifies the details of a user: all the cached pages that display the information about this user in the front-office have to be cleared, but they are in another application.

What you should do is to setup an action in the front-office application that will behave like a web service. The back-office will call this action through HTTP, passing it the detail of the modification in the model. The web service will determine which pages need to be cleared according to the part of the model that was modified.

A tutorial will soon be published to describe this procedure.

Testing and monitoring the cache improvements

Staging environment

The caching system is prone to new errors in the production environment that can't be detected in the development environment, since the HTML cache is deactivated by default in development. If you use the HTML cache, it is strongly recommended to add a new environment, that will be called `staging` here, which is a copy of the production environment (thus with cache activated) but with the web debug set to on.

To set it up, edit the `settings.yml` of your application and add at the top:

```
staging:
  .settings:
    web_debug:              on
    cache:                  on
    no_script_name:         off
```

In addition, create a new front controller by copying the production one (probably named `index.php` in the `myproject/web/` directory) to a new `myapp_staging.php` and edit it to change the `SF_ENVIRONMENT` value:

```
define('SF_ENVIRONMENT', 'staging');
```

That's it, you have a new environment. Use it by adding the front controller name after the domain name:

```
http://myapp.example.com/myapp_staging.php/user/list
```

Monitor the performance improvement

You will notice the familiar web debug box at the top right corner of the browser window, showing that the cache is set to `on`.

After a bunch of flags, this debug box also displays the process duration. Go ahead, clear the cache and make some tests: the second time you request a page, depending on the complexity of the action and template, can be several times faster with caching turned on.

To get more details about the caching impact, activate the debug mode by editing the `myapp_staging.php` and changing the `SF_DEBUG` value:

```
define('SF_DEBUG', true);
```

Don't forget to clear the cache before requesting the page again.

Beware that the debug mode greatly decreases the speed of your application, since a lot of information is logged and made available to the web debug box. So the processed time in debug mode is not representative of what it will be when the debug mode is turned off.

On the other hand, the web debug mode allows you to get additional information: the number of database requests needed to display the page, the ability to reload the page without caching (the middle button in the top part of the debug box) and full detail about the events encountered by the framework objects (the information bubble in the top part of the box).

Identifying cache parts

In pages that contain cached parts (slots, page or fragments), the web_debug mode allows you to display information about each part. By clicking on the 'cache information' link, you open a detail box showing:

- the internal URI of the fragment,
- the lifetime of the fragment,
- the moment when it was last modified

This will help you identify problems when dealing with out-of-context fragments, to see when the fragment was created and which parts of a template you can actually cache.

HTTP 1.1 and client-side caching

The HTTP 1.1 protocol defines a bunch of headers that can be of great use to speed further up an application by piloting the browser's cache system.

The HTTP 1.1 specifications of the W3C describe in detail these new headers. If a page is using the 'page' type cache in symfony, it can use one or more of the following mechanisms.

ETags

When `ETag` is activated, the web server adds to the response a special header containing a md5 hash of the response itself.

```
ETag: "1A2Z3E4R5T6Y7U"
```

The user's browser will store this hash, and send it again together with the request the next time it needs the same page. If the new hash shows that the page didn't change since the first request, the browser doesn't sent the response back. Instead, it just sends a '304: not modified' header. It saves cpu time (gzipping) and bandwidth (page transfer) for the server, and cpu time (page rendering) for the client. Overall, pages in cache with an etag are even faster to load than pages in cache without etag.

In symfony, you activate the etags feature for the whole application in the `settings.yml`. Here is the default setting:

```
all:
  .settings:
    etag: on
```

When the server receives a request for a page containing an etag, it processes this page as it would usually. For a page of 'page' type cache, the response it directly taken from the cache. The server will compute a new md5 hash of the cached response and see that it's the same as the one sent by the browser. Instead of sending the response again, the server will send a 304 header only, and the browser will redisplay the page it keeps in its local cache.

Conditional GET

When the server sends the response to the browser, it can add a special header to specify when the data contained in the page was last changed:

```
Last-Modified: Sat, 23 Nov 2005 13:27:31 GMT
```

When the browser needs the page again, it adds to the request

```
If-Modified-Since: Sat, 23 Nov 2005 13:27:31 GMT
```

The server can then compare the value kept by the client and the one returned by its application. If they match, the server returns a '304: not modified' header, saving bandwidth and cpu time just like above.

In symfony, you can set the `last_modified` response header just like you would for another header. For instance, in an action:

```
$this->getResponse()->setHttpHeader('Last-Modified', $date);
```

This date can be the actual date of last update of the data used in the page, given from your database or your file system. If you use a 'page' type cache, you just need to set the `last_modified` header to the current `time()`.

Vary

Another HTTP 1.1 header is `Vary`. It defines which parameters a page depends on, and is used by browsers to build cache keys. For example, if the content of a page depends on cookies, you can set its `Vary` header as follows:

```
Vary: Cookie
```

Most often, it is difficult to set the cache type to 'page' in symfony because the page may vary according to the cookie, the user language, or something else. If you don't mind expanding the size of your cache, you can use the 'page' type in these cases, providing you set the `Vary` header properly. This can be done for the whole application (for instance in a filter) or in a per action basis, using the `Response` related method. For instance, from an action:

```
$this->getResponse()->addVaryHttpHeader('Cookie');
$this->getResponse()->addVaryHttpHeader('User-Agent');
$this->getResponse()->addVaryHttpHeader('Accept-Language');
```

Symfony will store a different version of the page in the cache for each value of these parameters. This will increse the size of the cache, but whenever a request matching these headers is received by the server, it is taken from the cache instead of being processed. This is a great performance tool for pages that vary only according to request headers.

Cache-Control

Up to now, even by adding headers, the browser kept sending requests to the server even if it held a cached version of the page. There is a way to avoid that by adding `Cache-Control` and `Expires` headers to the response. These headers are deactivated by default in PHP, but symfony can override this behaviour to avoid unecessary requests to your server.

Beware that the major consequence of turning this mechanism on is that your server logs won't show all the requests issued by the users, but only the ones received. If the performance gets better, the apparent popularity of the site may decrease in the statistics.

As usual, it's by calling a methods of the `Response` object that you can trigger this behaviour. In an action, define the maximum time a page should be cached (in seconds) as follows:

```
$this->getResponse()->addCacheControlHttpHeader('max_age=60');
```

It also allows you to specify under which conditions a page may be cached, to avoid that providers cache keep a copy of private data (like bank account numbers):

```
$this->getResponse()->addCacheControlHttpHeader('private=True');
```

Using `Cache-Control` HTTP directives, you get the ability to fine tune the various cache maechanisms between your server and the client's browser. For a detailed review of the `Cache-Control` directives, see the Cache-Control specifications at W3C.

One last header is ignored by PHP but can be set through symfony: the `Expires` header:

```
$this->getResponse()->setHttpHeader('Expires', $date);
```

When to use HTTP 1.1 cache?

If there is a slight chance that some of the browsers of your website's users may not support HTTP 1.1, there is no risk when activating the HTTP 1.1 cache features. A browser receiving headers that it doesn't understand simply ignores it, so you are advised to setup the HTTP 1.1 cache mechanisms whenever your web server supports them.

In addition, HTTP 1.1 headers are also understood by proxies and caching servers. Even if a user's browser doesn't understand it, there will probably be a device in the route of the request to take advantage of it.

Postscript

In addition to the HTML cache, symfony has two other cache mechanisms, which are completely automated and transparent to the developer. In the production environment, the *configuration* and the *template translations* are cached in files stored in the `myproject/cache/config/` and `myproject/cache/i18n/` directories without any intervention. This already speeds up the delivery of pages a lot, but the most powerful feature, the HTML cache, can not be fully automated. That's why it relies on configuration and additional code.

One last word about speeding up PHP applications: Eaccelerator also increases performance of symfony PHP scripts by caching them in compiled state, so that the overhead of compiling is almost completely eliminated. This is particularly true for the Propel libraries, which contain a great amount of code. Eaccelarator is compatible with symfony, and their effects can be combined.

How to debug a project

Overview

No matter how good a coder you are, you will face errors one day or another - even if you use symfony. Detecting and understanding errors is one key of fast application development. Fortunately, symfony provides several debug tools for the developer.

Debug modes

Symfony debug mode

Symfony has a debug mode that facilitates the development and the debug. When it is on, the configuration is parsed at each request, so a change in the configuration has an immediate effect, without any need to clear the `config` cache folder. In order to prevent the production environment from being able to ignore any existing HTML cache this ability is only available in debug mode. (You can learn more about the cache feature in the related chapter). Eventually, the symfony debug mode activates the Propel debug mode, so any error in a call to a propel object will display a detailed chain of calls.

As the debug mode has to be set very early in the process of answering a request, it is not defined in a YAML configuration file but in the front controller. For instance, if your main application is called `myapp`, you probably have in the `myproject/web` two PHP scripts with a different debug mode configuration:

```
// in index.php, the production front controller,
// the debug mode is deactivated
define('SF_DEBUG', false);
//in myapp_dev.php, the development front controller,
// the debug mode is activated
define('SF_DEBUG', true);
```

With this configuration, not only will the production environment be faster, but it will also be protected from a few debug features (including the web debug toolbar, see below) that must not be given to end users for the sake of the application.

XDebug mode

The XDebug extension allows you to extend the amount of information that is logged by the web server. Symfony integrates the XDebug messages in its own debug feedback, so it is a good idea to activate it when you debug the application. Unfortunately, the extension installation - and configuration - depends very much on your platform, so you need to activate/deactivate it manually in your `php.ini` file after installation.

An example configuration for this extension could be (*nix platform):

```
;...
zend_extension="/usr/local/lib/php/extensions/no-debug-non-zts-20041030/xdebug.so"
;xdebug.profiler_enable=1
;xdebug.profiler_output_dir="/tmp/xdebug"
xdebug.auto_trace=On            ; enable tracing
```

```
xdebug.trace_format=0
;xdebug.show_mem_delta=0        ; memory difference
;xdebug.show_local_vars=1
;xdebug.max_nesting_level=100
```

In a Windows installation, you just need to add the following line to the Dynamic Extensions section:

```
extension=php_xdebug.dll
```

You have to restart you web server for the XDebug mode being activated.

Don't forget to deactivate the XDebug mode in your production platform.

Log files

The only way to understand what went wrong during the execution of a request is to have information about the various steps of the execution. Fortunately, your web server and symfony have the good habit of logging lots of data that can be used for debugging into files.

Web server log level configuration

First of all, a little reminder: PHP has a parameter `error_reporting`, defined in the `php.ini`, that defines which events are logged. Symfony allows you to override this value for each application by setting a specific `error_reporting` in the `settings.yml` file:

```
prod:
 .settings:
    error_reporting:  257

dev:
  settings:
    error_reporting:  4095
```

The numbers are a short way of writing error levels (refer to the PHP documentation on error reporting for more details about it). Basically, 4095 is a shortcut for `E_ALL | E_STRICT`, and 257 stands for `E_ERROR | E_USER_ERROR` (the default value for every new environment).

In the production environment to avoid execution slow-downs, only the critical PHP errors are logged by the server; but in the development environment, all the types of event are logged so that the developer can have all the necessary information to trace errors.

The location of the server log files depends on your installation, but they are often in a directory called `logs` in the web server tree structure.

Symfony log files

In addition to the classic PHP logs, symfony can trace a lot of custom events. These symfony logs can all be found under the `myproject/log/` directory. There is one file per application and per environment; the development environment log file of the `myapp` application is named `fo_dev.log`, the production one is named `myapp_prod.log`, and so on.

If you have a symfony application running, take a look at its log files. The syntax is very simple: for every event, one line is added to the log file of the application, where the exact time of the event, the nature of the event, the object being processed and the details of the process are written. Here is an example:

```
Nov 15 16:30:25 symfony [info] {sfActions} call "barActions->executemessages()"
Nov 15 16:30:25 symfony [debug] SELECT bd_message.ID, bd_message.SENDER_ID, bd_message.RECIPIENT_
Nov 15 16:30:25 symfony [info] {sfCreole} executeQuery(): SELECT bd_message.ID, bd_message.SENDER
Nov 15 16:30:25 symfony [info] {sfRenderView} set slot "leftbar" (bar/index)
Nov 15 16:30:25 symfony [info] {sfRenderView} set slot "messageblock" (bar/messages)
Nov 15 16:30:25 symfony [info] {sfRenderView} execute view for template "messagesSuccess.php"
Nov 15 16:30:25 symfony [info] {sfPHPView} render "/home/production/myproject/fo/modules/bar/temp
Nov 15 16:30:25 symfony [info] {sfPHPView} render to client
Nov 15 16:30:25 symfony [error] {sfActionStopException}
```

Lots of information can be found in these files, including the actual SQL queries sent to the database, the templates called, the chain of calls between objects, and so on. You can easily add your own logs, as explained later in this chapter.

By the way, don't forget to periodically purge the `log/` directory of your applications, because theses files have the strange habit of growing by several megabytes in a few days - depending, of course, on your traffic.

Symfony log level configuration

There are 8 levels of log messages: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` and `debug`, which are the classical levels of the [PEAR_Log package][3]. You can configure the maximum level to be logged in each environment in the `logging.yml` configuration file of each application. Here is the content of the default configuration:

```
prod:
  level:  err

dev:

test:

all:
#  active: on
#  level:  debug
```

This means that the symfony logging mechanism is active by default in all environments (but can be deactivated). In all environments except the production one, all the messages are logged (up to the least important level - the `debug` level). In production, only the most important messages are logged. You can change the `level` in this file for each environment to limit the type of logged messages.

The value of these parameters is accessible during execution through `sfConfig::get('sf_logging_active')` and `sfConfig::get('sf_logging_level')`.

Web debug

The log files contain interesting information, but are not very easy to read. The most basic task, which is to find the lines logged for a particular page, can be quite tricky if you have several users working with an

application. That's when you start to need a web debug toolbar.

Main configuration

To activate the web debug toolbar for an application, open the `settings.yml` and look for the `web_debug` key. Its default value is `off`, so you need to activate it manually if you want it - except in the `dev` environment where the default configuration has it set to `on`:

```
dev:
  .settings:
    web_debug:              on
```

Next time you display a page in an environment with web debug toolbar enabled, you will see a semi-transparent box on the top right corner of the window, showing the value of a few important settings:

To hide it, press the red cross button on the top right corner.

You can deactivate the web debug toolbar manually from within an action by simply changing this setting:

```
sfConfig::set('sf_web_debug', false);
```

Log messages

All the messages logged in the symfony log file for the current request are accessible with the web debug toolbar: just press the information bubble to display a list of the messages:

The top bar contains filter links that allow you to toggle the visibility of messages of a certain type. By default, all the messages are visible, so clicking on the 'sfActions' link will hide the `sfActions`-labeled messages.

The 'ms' column displays the time elapsed between each message, which is roughly equivalent to the execution time of the related method call. This should help you to find the longest sequences of each request - and optimize it.

The red cross closes the log messages table.

Log messages in XDebug mode

In XDebug mode, the log messages are much richer. Each line of the log messages table has a double-arrow button which, when pressed, reveals further details about the related request:

All the PHP script files and the functions that are called are logged in XDebug mode, and symfony knows how to link this information with its internal log. If something goes wrong, the XDebug mode gives you the maximum detail to find out why.

Manual debugging

Getting access to the framework debug messages is good, but being able to log your own is better. Symfony provides shortcuts, accessible from both actions and templates, to help you trace events and/or values during request execution.

Adding a log message

You can manually add a message in the log file (and, consequently, to the log messages table):

```
// in an action
$this->logMessage($message, $level);
// in a template
<?php echo log_message($message, $level) ?>
```

If you use the template version, don't forget to declare the `Debug` helper with `use_helper('Debug')`. The `$level` can have the same values as in the log messages (`emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` and `debug`). For instance, if your `indexSuccess.php` template of the `main` module looks like:

```
<?php use_helper('Debug') ?>
...
<?php if ($problem): ?>
  <?php echo log_message('{mainActions} been there', 'err') ?>
  ...
<?php endif ?>
```

...then if you have a `$problem` the web debug toolbar will show:

The red color of the header shows that at least one message of `err` level was raised. Click on the information bubble icon to display the messages. The list will contain:

Alternatively, to write a message in the log from anywhere in your application, use the following syntax:

```
sfContext::getInstance()->getLogger()->info($message);
sfContext::getInstance()->getLogger()->err($message);
...
```

Debug short message

If you don't want to add a line to the debug log, but just trace an short message or a value, you should use `debug_message` instead of `log_message`. The message will appear directly at the bottom of the web debug toolbar.

```
// in an action
$this->debugMessage($message);
// in a template
<?php echo debug_message($message) ?>
```

For instance, for a template call:

```
<?php use_helper('Debug') ?>
...
<?php if ($problem): ?>
  <?php echo debug_message('been there') ?>
  ...
<?php endif ?>
```

...will display:

[3]: http://pear.php.net/package/Log/ "PEAR Log package'

How to populate a database

Overview

In the process of application development, developers are often faced to the problem of database population. If a few specific solutions exist for some database system, none can be used on top of the object relational mapping. Thanks to YAML and the `sfPropelData` object, symfony facilitates the batch processes that take data from a text source to a database.

Introduction

Imagine a project where registered users are stored in a `User` table with the following columns:

User
id
name
login
hashed_password
email
created_at

The `User` Propel object contains standard accessors. For this example project, a `->setPassword()` method that sets the value of the `hashed_password` field from a password in clear is added to the `myproject/lib/model/User.class.php` class file:

```
class User extends BaseUser
{
  ...
  public function setPassword($password)
  {
    $this->setHashedPassword(md5(password));
  }
}
```

This method stores a MD5 hash of its argument, to avoid storing in clear a secret piece of data.

The `id` column is auto incremented.

If you want to enter some records in this table, for unit tests or in order to reinitialize the database of the test environment every day, you need to be able to use a text data file and to import it into the database.

Data file syntax

Symfony can read data files that follow a very simple YAML syntax. Under a header that defines the class of the records to be put in the database, a list of record is written, each record labeled by a unique string. For each record, a set set of `fieldname:value` is listed. For instance:

```
User:
  bob_walter:
    name:       Bob Walter
    login:      bwalter
    password:   sarah34
    email:      bob.walter@foomail.com

  peter_clive:
    name:       Peter Clive
    login:      pclive
    password:   gurzikso
    email:      peter.clive@foomail.com
```

The keys used in this file will be Camelized to find the appropriate setter (`->setName()`, `->setLogin()`, `->setPassword()`, `->setEmail()`). This means that it is possible to define a `password` key even if the actual table doesn't have a `password` field because the `->setPassword()` method exists in the `User` object.

The column `id` doesn't need to be defined, since it is an auto-increment field: the database layer knows how to determine it.

The `created_at` column isn't defined either, because symfony knows that fields named `created_at` have to be set to the current system time when created. This would be the same with an `updated_at` field.

Save this data in an `import_data.yml` file, in the `myproject/data/fixtures/` directory.

Import Batch

Initial declarations

An import batch is a PHP script that reads the data from YAML files and transfers it to the database. Since it has to access the Propel classes and use the symfony configuration, such a script must start just like any front controller:

```
<?php

define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/..'));
define('SF_APP',         'myapp');
define('SF_ENVIRONMENT', 'dev');
define('SF_DEBUG',       true);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATO

sfContext::getInstance();
```

The definition of an application (`myapp`) and of an environment (`dev`) allows symfony to determine a configuration to be loaded. Once executed, this code allows the batch to access Propel objects, and to autoload any symfony class.

`sfPropelData` call

The `sfPropelData` is an object who knows how to read a YAML data file and use it to add records to a Propel-enabled database. It is very easy to use: you just need to call its `->loadData()` method, passing as an argument a file or a directory that contains the data to be loaded into the database.

Create a `load_data.php` script in the `myproject/batch/` directory starting with the previous lines, plus the following ones:

```
$data = new sfPropelData();
$data->loadData(sfConfig::get('sf_data_dir').DIRECTORY_SEPARATOR.'fixtures');

 ?>
```

To launch the script, just type in a command line:

```
$ cd myproject/batch
$ php load_data.php
```

This batch will put the two records labeled `bob_walter` and `peter_clive` in the database - provided that the database connection settings are properly entered in the `myproject/apps/myapp/config/databases.yml` configuration file.

> **Note**: If you added a database connection or changed the default connection name (`'propel'`) in your `databases.yml`, you will need to specify this connection name when you call the `->loadData()` method. Use the second argument for that. For instance, if you want to populate a database using a connection named `symfony`, write:
>
> ```
> [php]
>  $data->loadData(sfConfig::get('sf_data_dir').DIRECTORY_SEPARATOR.'fixtures', 'symfony'
> ```

Add or replace

By default, the `->loadData()` method will not only add the records from the YAML files given as parameters, it will also erase the existing records in the tables that are modified. To change this behavior, you need to set the `deleteCurrentData` property to false before calling the `->loadData()` method:

```
$data->setDeleteCurrentData(false);
```

To get the current value of the `deleteCurrentData` property, use the associated getter:

```
$status = $data->getDeleteCurrentData(false);
```

Linked tables

Adding records in one table is easy, but the things could get tricky with linked tables. Imagine that the users of your website can write posts in a blog-like interface. Your model would also have a `Post` table with more or less the following fields:

Post

id

user_id

title

body

created_at

updated_at

The problem is: how can you define the value of the `user_id` field if you have no idea of the `id` given automatically to the records of the `User` table?

This is where the labels given to the records become really useful. To add a post written by Bob Walter, you simply need to add to the `myproject/data/fixtures/import_data.yml` data file:

```
Post:
  post01:
    user_id:      bob_walter
    title:        Today is the first day of the rest of my life
    body:         I have plenty to say, but I prefer to stay short. Enjoy.
```

The `sfPropelData` object will recognize the label that you gave to a user previously in the `import_data.yml`, and grab the primary key of the corresponding User record to set the `user_id` field. You don't even see the ids of the records, you just link them by their labels: it can't be simpler.

The only constraint to a link file is that the objects called in a foreign key have to be defined earlier in the file - that is, as you would do if you defined them one by one. The data files are parsed from the top to the bottom, and the order in which the records are written is then important.

Flat file vs. separated files

One data file can contain declarations of several classes. For instance, consider the following data file:

```
User:
  bob_walter:
    name:         Bob Walter
    login:        bwalter
    password:     sarah34
    email:        bob.walter@foomail.com

  peter_clive:
    name:         Peter Clive
    login:        pclive
    password:     gurzikso
    email:        peter.clive@foomail.com

Post:
  test_post:
    user_id:      bob_walter
    title:        Today is the first day of the rest of my life
    body:         I have plenty to say, but I prefer to stay short. Enjoy.

  another_test_post:
```

```
    user_id:      bob_walter
    title:        I don't know what to say today
    body:         As my previous post was so briliant, I find myself a little dry.

Comment:
  enthusiast_comment:
    user_id:      peter_clive
    post_id:      test_post
    body:         Hey Bob, I'm so glad you finally found something interesting to say.

  disappointed_comment:
    user_id:      peter_clive
    post_id:      another_test_post
    body:         Mate, you really disappoint me. I expected much more or your great potential.
```

It is getting quite long. Symfony allows you to cut the file in pieces, that will be parsed in the alphabetical order of their file names. For instance, you could separate this file into three ones, one for each class. To make sure that they are parsed in the correct order, prefix them with an ordinal number:

```
100_user_import_data.yml
200_post_import_data.yml
300_comment_import_data.yml
```

The same `->loadData()` method call still works, since the parameter was a directory:

```
$data->loadData(sfConfig::get('sf_data_dir').DIRECTORY_SEPARATOR.'fixtures');
```

You can, if you wish, use only one of these files for the batch:

```
$data->loadData(sfConfig::get('sf_data_dir').DIRECTORY_SEPARATOR.'fixtures'.DIRECTORY_SEPARATOR.'
```

Alternate YAML syntax

YAML has an alternate syntax for associative arrays that may make your data files more readable, especially for the tables that contain many foreign keys:

```
Comments:
  c123:
    user_id:    u65
    post_id:    p23
    body:       first blabla

  c456:
    user_id:    u97
    post_id:    p64
    body:       second blabla
```

The same can be written in a shorter way:

```
Comments:
  c123: { user_id: u65, post_id: p23, body: first blabla }
  c456: { user_id: u97, post_id: p64, body: second blabla }
```

For more information about the YAML syntax, refer to the YAML website

How to do unit testing

Overview

Unit tests are one of the greatest advances in programming since object orientation. They allow for a safe development process, refactoring without fear, and can sometimes replace documentation since they illustrate quite clearly what an application is supposed to do. Unit testing can also be used to avoid regression. Refactoring a method can create new bugs that didn't use to appear before. That's why it is also a good practice to run all unit tests before deploying a new realease of an application in production - this is called regression testing. Symfony supports and recommends unit testing, and provides tools for that.

There is no perfect solution for unit testing PHP applications built with symfony. This chapter decribes three solutions, each answering the need only partially. If you have an extensive approach to unit testing, you will probably need to use all the three.

Simple test

There are many unit test frameworks in the PHP world, mostly based on Junit. Symfony integrates the most mature of them all, Simple Test. It is stable, well documented, and offers tons of features that are of considerable value for all PHP projects, including symfony ones. If you don't know it already, you are strongly advised to browse their documentation, which is very clear and progressive.

Simple Test is not bundled with symfony, but very simple to install. First, download the Simple Test PEAR installable archive at SourceForge. Install it via pear by calling:

```
$ pear install simpletest_1.0.0.tgz
```

If you want to write a batch script that uses the Simple Test library, all you have to do is insert these few lines of code on top of the script:

```
<?php
require_once('simpletest/unit_tester.php');
require_once('simpletest/reporter.php');

?>
```

Symfony does it for you if you use the test command line; we will talk about it shortly.

> **Note**: Due to non backward-compatible changes in PHP 5.0.5, Simple Test is currently not working if you have a PHP version higher than 5.0.4. This should change shortly (an alpha version addressing this problem is available), but unfortunately the rest of this tutorial will probably not work if you have a later version.

Unit tests in a symfony project

Default unit tests

Each symfony project has a `test/` directory, divided into application subdirectories. If you generated a scaffolding, you might already find a few tests in files labeled like `myproject/test/myapp/mymoduleActionsTest.php`:

```
<?php

class mymoduleActionsWebBrowserTest extends UnitTestCase
{
  private
    $browser = null;

  public function setUp ()
  {
    // create a new test browser
    $this->browser = new sfTestBrowser();
    $this->browser->initialize('hostname');
  }

  public function tearDown ()
  {
    $this->browser->shutdown();
  }

  public function test_simple()
  {
    $url = '/mymodule/index';
    $html = $this->browser->get($url);
    $this->assertWantedPattern('/mymodule/', $html);
  }
}

?>
```

The `UnitTestCase` class is the core class of the Simple Test unit tests. The `setUp()` method is run just before each test method, and `tearDown()` is run just after each test method. The actual test methods start with the word 'test'. To check if a piece of code is behaving as you expect, you use an assertion, which is a method call that verifies that something is true. In Simple Test, assertions start by `assert`. In this example, one unit test is implemented, and it looks for the word 'user' in the default page of the module. This autogenerated file is a stub for you to start.

As a matter of fact, every time you call a `symfony init-module`, symfony creates a skeleton like this one in the `test/[appname]/` directory to store the unit tests related to the created module. The trouble is that as soon as you modify the default template, the stub tests don't pass anymore (they check the default title of the page, which is 'module $modulename'). So for now, we will erase these files and work on our own test cases.

Add a unit test

Let's imagine that you need to create a `Util` class that normalizes a string by removing all special characters. Before even writing the class, write the unit test. For that, add a couple of test cases that illustrate extensively what you expect of the class in a `UtilTest.php` file (all the test case files must end with `Test` for Simple

Test to find them):

```
<?php

require_once('Util.class.php');

class TagTest extends UnitTestCase
{
  public function test_normalize()
  {
    $tests = array(
      'FOO'       => 'foo',
      '   foo'    => 'foo',
      'foo  '     => 'foo',
      ' foo '     => 'foo',
      'foo-bar'   => 'foobar',
    );

    foreach ($tests as $string => $normalized_string)
    {
      $this->assertEqual($normalized_string, Util::normalize($string));
    }
  }
}

?>
```

> **Note**: As a good practice, we recommend that you name the test files using the class they are supposed to test, and the test cases using the methods they are supposed to test. Your `test/` directory will soon contain a lot of files, and finding a test might prove difficult in the long run if you don't.

Unit tests are supposed to test one case at a time, so we decompose the expected result of the text method into elementary cases. We want the `Util::normalize()` method to return a lower-case version of its argument, without any spaces - either before or after the argument - and without any special characters. The five test cases defined in the `$test` array are enough to test that.

For each of the elementary test cases, we then compare the normalized version of the input with the expected result, with a call to the `->assertEqual()` method. This is the heart of a unit test. If it fails, the name of the test case will be output when the test suite is run. If it passes, it will simply add to the number of passed tests.

We could add a last test with the word `' FOo-bar '`, but it mixes elementary cases. If this test fails, you won't have a clear idea of the precise cause of the problem, and you will need to investigate further. Keeping to elementary cases gives you the insurance that the error will be located easily.

> **Note**: The extensive list of the `assert` methods can be found in the Simple Test documentation.

Unit tests accessing the database

If you want to include unit tests which need a connection to the database, initialize it in the `setUp()` method as follows:

```
public function setUp ()
{
  // initialize the database manager
  $databaseManager = new sfDatabaseManager();
  $databaseManager->initialize();
}
```

You can then use the database connections and Propel objects just like in actions.

Running unit tests

The symfony command line allows you to run all the tests at once with a single command (remember to call it from your project root directory):

```
$ symfony test myapp
```

Calling this command executes all the tests of the `test/myapp/` directory, and for now it is only those in our new `UtilTest.php` set. These tests will not pass and the command line will show:

```
Warning: main(Util.class.php): failed to open stream: No such file or directory in WEB_DIR/sf_san
```

This is normal: the `Util` class doesn't exist yet. Create the `Util.class.php` now in the `lib/` directory:

```
class Util
{
  public static function normalize($string)
  {
    $n_string = strtolower($string);

    // remove all unwanted chars
    $n_string = preg_replace('/[^a-zA-Z0-9]/', '', $n_string);

    return trim($n_string);
  }
}
```

Launch the tests again. This time they will pass, and the command line will show:

```
$ symfony test myapp
Test suite in (test/myapp)
OK
Test cases run: 1/1, Passes: 5, Failures: 0, Exceptions: 0
```

> **Note**: Tests launched by the symfony command line don't need to include the Simple Test library (`unit_tester.php` and `reporter.php` are included automatically).

Test driven development

The greatest benefit of unit tests is experienced when doing test-driven development. In this methodology, the tests are written before the function.

With the example above, you would write an empty `Util::normalize()` method, then write the first test

case ('Foo'/'foo'), then run the test suite. The test would fail. You would then add the necessary code to transform the argument into lowercase and return it in the `Util::normalize()` method, then run the test again. The test would pass this time.

So you would add the tests for blanks, run them, see that they fail, add the code to remove the blanks, run the tests again, see that they pass. Then do the same for the special characters.

Writing tests first helps you to focus on the things that a function should do before actually developing it. It's a good practice that others methodologies, like eXtreme Programming, recommend as well. Plus it takes into account the undeniable fact that if you don't write unit tests first, you never write them.

One last recommendation: keep your unit tests as simple as the ones described here. An application built with a test driven methodology ends up with roughly as much test code as actual code, so you don't want to spend time debugging your tests cases...

Simulating a web browsing session

Web applications are not all about objects that behave more or less like functions. The complex mechanisms of page request, HTML result and browser interactions require more than what's been exposed before to build a complete set of unit tests for a symfony web app.

We will examine three different ways to implement a simple web app test. The test has to do a request to a `mymodule/index` page, and assume that some "mytext" text. We will put this test into a `mymoduleTest.php` file, located in the `myproject/test/myapp/` directory.

The `sfTestBrowser` object

Symfony provides an object called `sfTestBrowser`, which allows your test to simulate browsing without a browser and, more important, without a web server. Being inside the framework allows this object to bypass completely the http transport layer. This means that the browsing simulated by the `sfTestBrowser` is fast, and independent of the server configuration, since it does not use it.

Let's see how to do a request for a page with this object:

```
$browser = new sfTestBrowser();
$browser->initialize();
$html = $browser->get('uri');

// do some test on $html

$browser->shutdown();
```

The `get()` request takes a routed URI as a parameter (not an internal URI), and returns a raw HTML page (a string). You can then proceed to all kinds of tests on this page, using the `assert*()` methods of the `UnitTestCase` object.

You can pass parameters to your call as you would in the URL bar of your browser:

```
$html = $browser->get('/myapp_test.php/mymodule/index');
```

The reason why we use a specific front controller (`myapp_test.php`) will be explained in the next section.

The `sfTestBrowser` simulates a cookie. This means that with a single `sfTestBrowser` object, you can request several pages one after the other, and they will be considered as part of a single session by the framework. In addition, the fact that `sfTestBrowser` uses routed URIs instead of internal URIs allows you to test the routing engine.

To implement our web test, the `test_Index()` method must be built as follows:

```
class mymoduleTest extends UnitTestCase
{
  public function test_Index()
  {
    $browser = new sfTestBrowser();
    $browser->initialize();
    $html = $browser->get('/myapp_test.php/mymodule/index');
    $this->assertWantedPattern('/mytext/', $html);
    $browser->shutdown();
  }
}
```

Since almost all the web unit tests will need a new `sfTestBrowser` to be initialized and closed after the test, you'd better move part of the code to the `->setUp()` and `->tearDown()` methods:

```
class mymoduleTest extends UnitTestCase
{
  private $browser = null;

  public function setUp()
  {
    $this->browser = new sfTestBrowser();
    $this->browser->initialize();
  }

  public function tearDown()
  {
    $this->browser->shutdown();
  }

  public function test_Index()
  {
    $html = $this->browser->get('/myapp_test.php/mymodule/index');
    $this->assertWantedPattern('/mytext/', $html);
  }
}
```

Now, every new `test` method that you add will have a clean `sfTestBrowser` object to start with. You may recognize here the auto-generated test cases mentioned at the beginning of this tutorial.

The `WebTestCase` object

Simple Test ships with a `WebTestCase` class, which includes facilities for navigation, content and cookie checks, and form handling. Tests extending this class allow you to simulate a browsing session with a http transport layer. Once again, the Simple Test documentation explains in detail how to use this class.

The tests built with `WebTestCase` are slower than the ones built with `sfTestBrowser`, since the web server is in the middle of every request. They also require that you have a working web server configuration. However, the `WebTestCase` object comes with numerous navigation methods on top of the `assert*()` ones. Using these methods, you can simulate a complex browsing session. Here is a subset of the `WebTestCase` navigation methods:

-	-	-
`get($url, $parameters)`	`setField($name, $value)`	`authenticate($name, $password)`
`post($url, $parameters)`	`clickSubmit($label)`	`restart()`
`back()`	`clickImage($label, $x, $y)`	`getCookie($name)`
`forward()`	`clickLink($label, $index)`	`ageCookies($interval)`

We could easily do the same test case as previously with a `WebTestCase`. Beware that you now need to enter full URIs, since they will be requested from the web server:

```
class mymoduleTest extends WebTestCase
{
  public function test_Index()
  {
    $this->get('http://myapp.example.com/myapp_test.php/mymodule/index');
    $this->assertWantedPattern('/mytext/');
  }
}
```

The additional methods of this object could help us test how a submitted form is handled, for instance to unit test a login process:

```
public function test_Login()
  {
    $this->get('http://myapp.example.com/myapp_test.php/');
    $this->assertLink('sign in/register');
    $this->clickLink('sign in/register');
    $this->assertWantedPattern('/nickname:/');
    $this->setField('nickname', 'testuser');
    $this->setField('password', 'testpwd');
    $this->clickSubmit('sign in');
    $this->assertWantedPattern('/test user logged in/');
  }
```

It is very handy to be able to set a value for fields and submit the form as you would do by hand. If you had to simulate that by doing a `POST` request (and this is possible by a call to `->post($uri, $parameters)`), you would have to write in the test function the target of the action and all the hidden fields, thus depending too much on the implementation. For more information about form tests with Simple Test, read the related chapter of the Simple Test documentation.

Selenium

The main drawback of both the `sfTestBrowser` and the `WebTestCase` tests is that they cannot simulate JavaScript. For very complex interactions, like with AJAX interactions for instance, you need to be able to reproduce exactly the mouse and keyboard inputs that a user would do. Usually, these tests are reproduced by hand, but they are very time consuming and prone to error.

The solution, this time, comes from the JavaScript world. It is called Selenium and is better when employed with the Selenium Recorder extension for Firefox. Selenium executes a set of action on a page just like a regular user would, using the current browser window.

Selenium is not bundled with symfony by default. To install it, you need to create a new `selenium/` directory in your `web/` directory, and unpack there the content of the Selenium archive. This is because Selenium relies on JavaScript, and the security settings standard in most browsers wouldn't allow it to run unless it is available on the same host and port as your application.

> **Note**: Beware not to transfer the `selenium/` directory to your production host, since it would be accessible from the outside.

Selenium tests are written in HTML and stored in the `selenium/tests/` directory. For instance, to do the simple unit test mentioned above, create the following file called `testIndex.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
  <title>Index tests</title>
</head>
<body>
<table cellspacing="0">
<tbody>
  <tr><td colspan="3">First step</td></tr>

  <tr>
    <td>open</td>
    <td>/myapp_test.php/</td>
    <td> </td>
  </tr>

  <tr>
    <td>clickAndWait</td>
    <td>link=go to mymodule</td>
    <td> </td>
  </tr>

  <tr>
    <td>assertTextPresent</td>
    <td>mytext</td>
    <td> </td>
  </tr>

</tbody>
</table>
</body>
```

```
</html>
```

A test-case is represented by an HTML document, containing a table with 3 columns: command, target, value. Not all commands take a value, however. In this case either leave the column blank or use a ` ` to make the table look better.

You also need to add this test to the global test suite by inserting a new line in the table of the `TestSuite.html` file, located in the same directory:

```
...
<tr><td><a href='./testIndex.html'>My First Test</a></td></tr>
...
```

To run the test, simply browse to

```
http://myapp.example.com/selenium/index.html
```

Select 'Main Test Suite', than click on the button to run all tests, and watch your browser as it reproduces the steps that you have told him to do.

> **Note**: As Selenium tests run in a real browser, they also allow you to test browser inconsistencies. Build your test with one browser, and test them on all the others on which your site is supposed to work with a single request.

The fact that Selenium tests are written in HTML could make the writing of Selenium tests a hassle. But thanks to the Firefox Selenium extension, all it takes to create a test is to execute the test once in a recorded session. While navigating in a recording session, you can add assert-type tests by right clicking in the browser window and selecting the appropriate check under the Append Selenium Command in the pop-up menu.

You can save the test to a HTML file to build a Test Suite for your application. The Firefox extension even allows you to run the Selenium tests that you have recorded with it.

> **Note**: Don't forget to reinitialize the test data before launching the Selenium test.

A few words about environments

Web tests have to use a front controller, and as such can use a specific environment (i.e. configuration). Symfony provides a `test` environment to every application by default, specifically for unit tests. You can define a custom set of settings for it in your application `config/` directory. The default configuration parameters are:

```
test:
  .settings:
    # E_ALL | E_STRICT & ~E_NOTICE = 2047
    error_reporting:        2047
    cache:                  off
    stats:                  off
    web_debug:              off
```

The cache, the stats and the web_debug toolbar are set to off. However, the code execution still leaves traces in a log file (`myproject/log/myapp_test.log`). You can have specific database connection settings, for instance to use another database with test data in it.

This is why all the external URIs mentioned above show a `myapp_test.php`: the `test` front controller has to be specified - otherwise, the default `index.php` production controller will be used in place, and you won't be able to use a different database or to have separate logs for your unit tests.

> **Note**: Web tests are not supposed to be launched in production. They are a developer tool, and as such, they should be run in the developer's computer, not in the host server.

The project control panel

Overview

The Command Line Interface provides useful commands to execute the usual tasks of an application development. However, symfony comes bundled with a web control panel for your projects, enabling you to achieve the same tasks as with the CLI from a web browser.

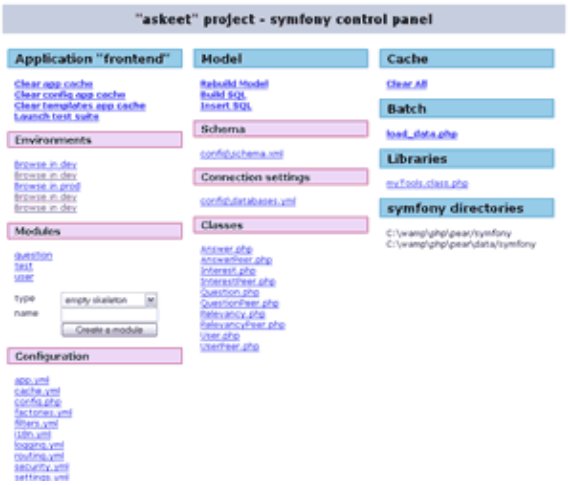
Installation

The project control panel is not installed by default in a project, but as it consists of a single file, it's pretty easy to add it to each project where you need it.

Locate the file called `sf_control_panel.php` in `$pear_data_dir/symfony/web/` and copy it to your project `web/` folder. That's it.

To test the control panel, just browse this script using your favorite web browser. As it's in the root web folder, it should be accessible with:

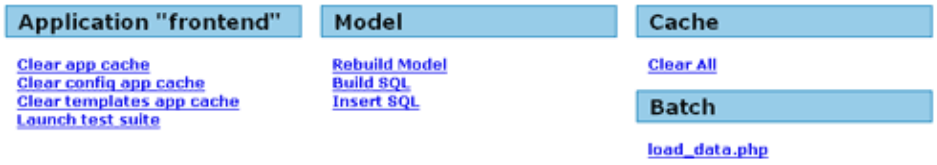
```
http://myapp.example.com/sf_control_panel.php
```



> **Note**: The execution of this script is restricted to the localhost for safety reasons. If the page returned is blank, it might be because you are calling a distant server. You can force the script to run in a distant server by removing lines 11 to 15, but beware of the security risks (see *Security* below).

Features

Tasks and batch



The control panel gives a web access to the usual CLI tasks for your application. You can easily clear the cache (for all the applications or selectively), rebuild the model or the database itself.

When you click on one of the task links (the bold links), the page displays again with a report of the task action. You can close the report at any time to keep on using the control panel.

The batch scripts located in the project `batch/` folder are also detected and can be executed from the control panel by a simple click.

> **Note**: Some of the symfony tasks can take some time to execute, according to the size of your project. If your web server has a small timeout configuration, you might end up with unfinished tasks. Increase your web server timeout to avoid these problems.

Browsing the project

The control panel automatically detects the applications and their environments. It provides links to browse each application in all the available environments.

Parsing the project

All the PHP, XML and YML files of your project can be parsed from within the control panel. It displays the project structure by separating the applications and the model.

For a given application, the list of existing modules is displayed. If you click on a module name, it reveals its structure and allows you to browse its actions, templates and configuration.

The source code is displayed with syntax highlighting, as long as you configured PHP with the appropriate option.

It can be very handy to check your data schema, the available methods of the model or the libraries included in your `lib/` folders.

Security

The script is configured to work only when called on localhost to avoid that if it accidentally gets transferred to production, nobody can access it and execute symfony tasks on your project.

This also means that you should exclude this file from your transfers to production. If you use the symfony sync task, this is done by adding the script to the `config/rsync_exclude.txt` file:

```
stats
.svn
web/uploads
cache
log
web/sf_control_panel.php
```

The script can execute symfony commands, and since those commands deal with the file system (creating, modifying and removing files), they may not work if the permissions given to your web server don't allow it. If the cache clearing commands should always work (simply because the cache files are created by your web server), the ones to init a module, a scaffolding or an administration will work only if your web server has write permissions to the `apps/` folder. This can be acceptable in a development environment, but it is extremely inadvisable for a production server. So once again, **do not use this control panel in the server used in production**.

How to deploy a project to production

Overview

Symfony offers shorthand commands to synchronize two versions of a website. They are mostly used to deploy a website from a development server to a final host, connected to the Internet.

Synchronization

Good practices

There are a lot of ways to synchronize two environments for a website. The basic file transfers can be achieved by an FTP connection, but there are two major drawbacks to this solution. First, it is not secure, the data stream transmits in the clear over the Internet and can be intercepted. Second, sending the root project directory by FTP is fine for the first transfer, but when you have to upload an update of your application, where only a few files changed, this is not a good and fast way to do it. Either you transfer the whole project again, which can be long, or you browse to the directories where you know that some files changed, and transfer only the ones with different modification dates. That's a long job, and it is prone to error. In addition, the website can be unavailable or buggy during the time of the transfer.

The solution that is supported by symfony is **rsync** synchronization through a **SSH** layer.

Rsync is a command line utility that provides fast incremental file transfer, and it's open source. By 'incremental', it means that only the modified data will be transferred. If a file didn't change, it won't be sent to the host. If a file changed only partially, only the differential will be sent. The major advantages is that rsync synchronizations transfer only a little data and are very fast.

Symfony adds SSH on top of rsync to secure the data transfer. More and more commercial hosts support an SSH tunnel to secure file uploads on their servers, and that's a good practice that symfony encourages.

For notes on installing rsync and SSH on Linux, read the instructions in the related websites. For Windows users, an open-source alternative called cwRsync exists, or you can try to install the binaries by hand (instructions can be found here). Of course, to be able to setup an SSH tunnel between an integration server and a host server, the SSH service has to be installed and running on *both* computers.

The `symfony sync` command

Doing a rsync over SSH requires several commands, and synchronization can occur a lot of times in the life of an application. Fortunately, symfony automates this process with just one command:

```
$ symfony sync production
```

This command, called from the root directory of a symfony project, launches the synchronization of the project code with the `production` hosted server. The connection details of this server are to be written in the project `properties.ini`, found in `myproject/config/`:

```
name=myproject
```

```
[production]
  host=myapp.example.com
  port=22
  user=myuser
  dir=/home/myaccount/myproject/
```

The connection settings will be used by the SSH client call enclosed in the symfony `sync` command line.

> **Note**: don't mix up the `production` environment (the host server, as defined in the `properties.ini` of the project) with the `prod` environment (the front controller and configuration used in production, as referred to in the configuration files of an application)

If you just call `symfony sync` like mentioned above, the rsync utility will run in dry mode by default (`--dry-run`), i. e. it will show you which files have to be synchronized but *without actually synchronizing them*. If you want the synchronization to be done, you have to mention it explicitly:

```
$ symfony sync production go
```

Ignore irrelevant files

If you synchronize your symfony project with a production host, there are a few files and directories that should not be transferred:

- All the `.svn` directories and their content: They contain source version control information, only necessary for development and integration
- `myproject/web/fronted_dev.php`: The web front controller for the development environment must not be available to the final users. The debugging and logging tools available when using the application through this front controller slow down the application, and give information about the core variables of your actions. It is something to keep off of the host server.
- The `cache/` and `log/` directories of a project must not be erased in the host server each time you do a synchronization. These directories must be ignored as well. If you have a `stats/` directory, it should probably be treated the same.
- The files uploaded by users: one of the good practices of symfony projects is to store the uploaded files in the `web/uploads/` directory. This allows us to exclude all these files by pointing to only one directory.

To exclude files from rsync synchronizations, open and edit the `rsync_exclude.txt` file under the `myproject/config/` directory. Each line can contain either a file, a directory, or a pattern:

```
.svn
cache
log
stats
web/uploads
web/myapp_dev.php
```

Thanks to the symfony file structure, you don't have too many files or directories to exclude manually from the synchronization. If you want to learn more about the way the files are organized in a symfony project, refer to the file structure chapter.

> **Note**: The `cache/` and `log/` directories must not be synchronized with the development server, but they must at least exist in the production server. Create them by hand if the `myproject/` project tree structure doesn't contain them.

Production server configuration

For your project to work in the production server, the symfony framework has to be installed in the host.

Install symfony in a production server

There are several ways to install symfony on a server, but they are not all adapted to a production environment. For instance, doing a PEAR install requires administrator rights on directories that might not be open to you if you share a web server.

Based on the principle that you will probably host several projects using symfony on the production web server, the recommended symfony installation is to uncompress the archive of the framework in a specific directory. Only the `lib/` and `data/` directories are necessary in a production server, so you can get rid of the other files (`bin/`, `doc/`, `test/` and the files from the root directory).

You should end up with a file structure looking like:

```
/home/myaccount/
  symfony/
    lib/
    data/
  myproject/
    apps/
    batch/
    cache/
    config/
    data/
    doc/
    lib/
    log/
    test/
    web/
```

For the `myproject` project to use the symfony classes, you have to set up two symbolic links between the application `lib/symfony` and `data/symfony`, and the related directories in the `symfony` installation:

```
$ cd /home/myaccount/myproject
$ ln -sf /home/myaccount/symfony/lib lib/symfony
$ ln -sf /home/myaccount/symfony/data data/symfony
```

Alternatively, if you don't have command line access, the files of the framework can be copied directly into the project `lib/` and `data/` directories:

```
copy /home/myaccount/symfony/lib/*   into /home/myaccount/myproject/lib/symfony
copy /home/myaccount/symfony/data/*  into /home/myaccount/myproject/data/symfony
```

Beware that in this case, each time you update the framework, you have to do it in all your projects.

For more information, all the possible ways to install symfony are described in the installation chapter.

Access to the symfony commands in production

While developing, you probably often use the following command:

```
$ symfony clear-cache
```

It is necessary, at least, each time you change the configuration or the object model of the project. When you upload a new version of your project in production, the cache also needs to be cleared if you want the application to work. You can easily do it by deleting the full content of the `myproject/cache/` directory (by ftp or with a ssh console). Alternatively, you can have the power of the symfony command line in the production host at the price of a slightly longer installation.

To use the command line, you need to install the pake utility. Pake is a PHP tool similar to the make command. It automates some administration tasks according to a specific configuration file called `pakefile.php`. The symfony command line uses the pake utility, and each time you type `symfony`, you actually call `pake` with a special `pakefile.php` located in the `symfony/bin/` directory (find more about pake in symfony in the project creation chapter). If you install symfony via PEAR, pake is installed as a requirement, so you usually don't see it at all and don't need to bother about what comes next. But if you do a manual installation, you have to uncompress the pake directory (get it from your symfony pear installation or download it from the pake website) into your directory in the production server. Just like for the symfony libs, you also have to add a symlink in order to enable symfony to use pake:

```
$ ln -sf /home/myaccount/pake/lib lib/pake
```

You should end up with something like this:

```
/home/myaccount/
  pake/
    lib/
  symfony/
    lib/
    data/
  myproject/
    apps/
    batch/
    cache/
    config/
    data/
      symfony/ -> /home/myaccount/symfony/data
    doc/
    lib/
      symfony/ -> /home/myaccount/symfony/lib
      pake     -> /home/myaccount/pake/data
    log/
    test/
    web/
```

To call the symfony command to do a clear-cache, you need to do:

```
$ cd /home/myaccount/myproject/
$ php lib/pake/bin/pake.php -f lib/symfony/data/symfony/bin/pakefile.php clear-cache
```

Alternatively, you can create a file called `symfony` in the `home/myaccount/myproject/` with:

```
#!/bin/sh

php lib/pake/bin/pake.php -f lib/symfony/data/symfony/bin/pakefile.php $@
```

Then, all you need to do in order to clear the cache is that good old

```
$ symfony clear-cache
```

Web command

If you want to have the power of the pake utility but without command line access, you can also create a web access for the clear-cache command.

For instance, you could save the following `webpake.php` in your `/home/myaccount/myproject/web/` directory:

```
<?php

// as we are in the web/ dir, we need to go up one level to get to the project root
chdir(dirname(__FILE__).DIRECTORY_SEPARATOR.'..');

include_once('/lib/symfony/pake/bin/pake.php');

$pake = pakeApp::get_instance();
try
{
  $ret = $pake->run('/data/symfony/bin/pakefile.php', 'clear-cache');
}
catch (pakeException $e)
{
  print "<strong>ERROR</strong>: ".$e->getMessage();
}

?>
```

Then, clearing the cache could be done by navigating to:

```
http://myapp.example.com/webpake.php
```

> **Note**: Beware that by letting web access to administration tools, you can compromise the safety of your website.

Upgrading your application

There will be times in the life of your project when you need to switch between two versions of an application. It can be in order to correct bugs, or to upload new features. You can also be faced with the problem of switching between two versions of the database. If you follow a few good practices, these actions will prove easy and harmless.

Show unavailability notice

Between the moment when you start the data transfer and the moment you clear the cache (if you modify the configuration or the data model), there are sometimes more than a few seconds of delay. You must plan to display an unavailability notice for users trying to browse the site at that very moment.

In the application `settings.yml`, define the `unavailable_module` and `unavailable_action` settings:

```
all:
  .settings:
    unavailable_module:     content
    unavailable_action:     unavailable
```

Create an empty `content/unavailable` action and a `unavailableSuccess.php` template:

```
// myproject/frontend/modules/content/actions/actions.class.php
public function executeUnavailable()
{
  $this->setTitle('website maintenance');
}

// myproject/frontend/modules/content/templates/unavailableSuccess.php
<h1>Site maintenance</h1>

<p>The website is currently being updated.</p>

<p>Please try again in a few minutes.</p>
```

Now each time that you want to make your application unavailable, just change the `available` setting:

```
all:

  .settings:

    available:              off
```

Don't forget that for a configuration change to be taken into account in production, you need to clear the cache.

> **Note**: The fact that the whole application can be turned off with only a single parameter is possible because symfony applications use a single entry point, the front web controller. You will find more information about it in the controller chapter.

Use two versions of your application

A good way to avoid unavailability is to have the project root folder configured as a symlink. For instance, imagine that you are currently using the version 123 of your application, and that you want to switch to the version 134. If your web server root is set to `/home/myaccount/myproject/web/` and that the production folder looks like that:

```
/home/myaccount/
```

```
  pake/
    lib/
  symfony/
    lib/
    data/
  myproject/     -> /home/production/myproject.123/
  myproject.123/
  myproject.134/
```

Then you can instantly switch between the two versions by changing the symlink:

```
$ ln -sf /home/myaccount/myproject/ /home/myaccount/myproject.134/
```

The users will see no interruption, and yet all the files used after the change of the symlink will be the ones of the new release. If, in addition, you emptied the `cache/` folder of your release 134, you don't even need to launch a clear-cache after switching applications.

Switch databases

You can extrapolate that technique to switching databases. Remember that the address of the database used by your application is defined in the `databases.yml` configuration file. If you create a copy of the database with a new name, say `myproject.134`, you just need to write in the `myproject.134/frontend/config/databases.yml`:

```
all:
  propel:
    class:        sfPropelDatabase
    param:
      datasource: symfony
      phptype:    mysql
      hostspec:   localhost
      database:   myproject.134
      username:   myuser
      password:   mypassword
      compat_assoc_lower: true
      compat_rtrim_string: true
```

As the `databases.yml` will be switched as the same time as the application itself, your application will instantly start querying the new database.

This technique is especially useful if your application has lots of traffic and if you can't afford any service interruption.

TODO

- Upgrading the data model
- Synchronization and source versioning
- reverting to a previous version

How to manage a user session

Overview

To store and retrieve data about the user session, use the `sfUser` class and its standard methods, or override it to encapsulate user logic into a custom `myUser` class.

Introduction

Information about the current user and his/her session can be found in the `sfUser` object.

Getting access to this object and its methods differs whether you are in an action:

```
$this->getUser();
```

or if you are in a template:

```
$sf_user
```

The `sfUser` class owns a few default attributes used by symfony to deal with classical user session interactions:

- `culture` holds the user Culture (language and country)
- authentication attributes (these are described in detail in the authentication and rigths chapter)

In addition, the `sfUser` class owns a parameter holder in which you can store custom attributes.

Custom user attributes

If you need to store information about a user until his/her session is closed, use the `->getAttribute()` and `->setAttribute()` methods. Attributes can store any type of data (strings, arrays, associative arrays, even objects).

For instance, to set and get an attribute `nickname` in an action:

```
$this->getUser()->setAttribute('nickname', 'foo');
...
$this->getUser()->getAttribute('nickname');
```

These informations are stored whatever the user, whether he/she is identified or not.

To check whether an attribute has been defined for a user, use the `->hasAttribute()` method.

```
$hasNickname = $this->getUser()->hasAttribute('nickname');
```

To remove an attribute from the session use:

```
$this->getUser()->getAttributeHolder()->remove('nickname');
```

> **Note**: If you need to store information just for the duration of one request - for instance to pass information through a chain of action calls - you may prefer the `sfRequest` class, which also has `->getAttribute()` and `->setAttribute()` methods.

Flash parameters

A recurrent problem with user attributes is the cleaning of the user session once the attribute is not needed anymore. Did you ever dream of an ephemeral attribute, one that you could define and forget, knowing that it will disappear after the next request and leave your user session clean for the future? Symfony provides a simple mechanism to keep the user session clean in this case: the **flash parameter**.

In your action, define the flash attribute like this:

```
$this->setFlash('attrib', $value);
```

The template will be built and decorated, the page will be delivered to the user, who will then make a new request to another action. In this second action, just get the value of the flash attribute by:

```
$value = $this->getFlash('attrib');
```

Then, forget about it. After delivering this second page, the `attrib` flash attribute will be cleared, erased, removed, in a word: flashed. And even if you don't require it during this second action, the flash will disappear from the session anyway.

If you need to access a flash attribute from a template, use the `$sf_flash` object:

```
<?php if($sf_flash->has('attrib')): ?>
  <?php $attrib = $sf_flash->get('attrib') ?>
<?php endif ?>
```

Flash parameters are a clean way of passing information to the next action, and once you start using them, you'll probably consider session attributes as an unforgivable archaism.

Extending the session class

When the application needs to hold key information in the session, it is better to encapsulate this logic into an extension of the `sfUser` class.

Let's imagine you are building a chat website where users have to choose a nickname. A simple way to store and read this nickname would be:

```
$this->getUser()->setAttribute('nickname', 'foo');
$this->getUser()->getAttribute('nickname');
```

Although this is exactly the previous example, this is not a very good solution, partly because it shows the real attribute and doesn't encapsulate the inner logic of the class. What would happen, for instance, if you decided suddenly to change the attribute name from 'nickname' to 'name' ? To settle this problem, you just need to

extend the `sfUser` class. Check the `myproject/apps/myapp/lib/` directory of your application: an empty `myUser` class just waits to be completed.

```
class myUser extends sfUser
{
  public function getNickname()
  {
    return $this->getAttribute('nickname');
  }

  public function setNickname($nickname)
  {
    return $this->setAttribute('nickname', $nickname);
  }
}
```

Then, in the action, the calls would be:

```
$this->getUser()->setNickname('foo');
$this->getUser()->getNickname();
```

The reason why symfony uses the custom `myUser` class instead of the prebuilt `sfUser` is that the `factories.yml` configuration file of the application specifies it. You could decide to write a completely new user class, with your own custom methods, and call it `myproject/apps/myapp/lib/myCustomUser.php`. But then, you would have to force symfony to use it instead of the `myUser` class by writing in the `factories.yml`:

```
all:
  user:
    class: myCustomUser
```

Session expiration

Session expiration occurs automatically after `sf_timeout` seconds. This constant is of 30 minutes by default and can be modified for each environment in the `myproject/apps/myapp/config/settings.yml` configuration file:

```
default:
  .settings:
    timeout:     1800
```

How to manage user credentials

Overview

Symfony offers simple mechanisms to identify a user, manage his credentials and restrict access to certain parts of your applications to authenticated users with certain credentials only.

User identification

The authenticated status of the user is set by the `->setAuthenticated()` method of a `sfUser/myUser` object. For instance, if you need to implement a simple version of user identification in a module called `myAccount` with two actions `login` and `logout`, you can write:

```
class myAccountActions extends sfActions
{
  public function executeLogin()
  {
    if ($this->getRequestParameter('login') == 'admin')
    {
      $this->getUser()->setAuthenticated(true);
    }
  }

  public function executeLogout()
  {
    $this->getUser()->setAuthenticated(false);
  }
}
```

User credentials

For authenticated users, symfony provides an array of credentials in the `sfUser` class that can be set via simple access methods `has`, `add`, `remove` and `clear`. Each credential can have any value.

```
$user = $this->getUser();
// add a credential
$user->addCredential($credential);
// add several credentials at once
$user->addCredentials($credential1, $credential2);
// check if the user has a credential
$user->hasCredential($credential)                  =>   true
// remove a credential
$user->removeCredential($credential);
$user->hasCredential($credential)                  =>   false
// remove all credentials
$user->clearCredentials();
$user->hasCredential($credential1)                 =>   false
```

Here is an example of credential definition:

```
class myAccountActions extends sfActions
{
```

```
  public function executeLogin()
  {
    if ($this->getRequestParameter('login') == 'superuser')
    {
      $user = $this->getUser();
      $user->setAuthenticated(true);
      $user->addCredential('superuser');
    }
    if ($this->getRequestParameter('login') == 'admin')
    {
      $user = $this->getUser();
      $user->setAuthenticated(true);
      $user->addCredentials('admin', 'superuser');
    }
  }

  public function executeRemoveSomeCredential()
  {
    $user = $this->getUser();
    if ($user->hasCredential('admin', 'superuser'))
    {
      $user->removeCredential('superuser');
    }
    if ($user->hasCredential('admin'))
    {
      $user->removeCredential('admin');
    }
  }

  public function executeLogout()
  {
    $user = $this->getUser();
    $user->clearCredentials();
    $user->setAuthenticated(false);
  }
}
```

Access restriction

Now that the user can be authenticated and given credentials, it is time to restrict access to some of your actions to grant access only to a subset of users.

This will be done with the `security.yml` module configuration file. This file can be found in the `myproject/apps/myapp/modules/mymodule/config/` (if it doesn't exist for your module, create it).

For example, to restrict the `read` action of the `myAccount` module to users with 'customer' credential, and the `update` action to the users with 'admin' or 'superuser' credentials, the `security.yml` file of the `myproject/apps/myapp/modules/myAccount/config/` directory will have to look like:

```
read:
  is_secure:   on
  credentials: customer

update:
  is_secure:   on
```

```
  credentials: [admin, superuser]

all:
  is_secure:  off
```

What happens when a user tries to access a restricted action depends on his credentials:

- If the user is identified and has the proper credentials, the action will be executed
- If the user is not identified, he/she will be redirected to the default `login` action (default/login). You can configure this action in the `myproject/apps/myapp/config/settings.yml` file.
- If the user is identified but doesn't have the proper credentials, he/she will be redirected to the default `secure` action (default/secure). You can configure this action in the `myproject/myapp/config/settings.yml` file.

The power of credentials with AND and OR

The YAML syntax used in the `security.yml` allows you to restrict access to users having a combination of credentials, using either AND-type or OR-type associations:

```
credentials: [ admin, superuser ]           ## admin AND superuser
credentials: [[ admin, superuser ]]         ## admin OR superuser
credentials: [[ admin, superuser ], owner] ## (admin OR superuser) AND owner
```

Session expiration

The session expiration is described in detail in the user session chapter.

ï»¿How to internationalize a project

Overview

Symfony has native internationalization automatisms that make the development of multilingual and locally adapted web applications a painless task.

Introduction

The internationalization (i18n) of an application covers three aspects:

- standards and formats (dates, amounts, numbers, etc.).
- text information contained in the database
- text translation (interface and content)

Symfony brings a solution to each of these issues.

User culture

The `sfUser` class, used to manage the user session, has a native implementation of the user language and country, which is called *culture*. Symfony provides a getter and a setter method for this attribute. Here is an example of their use in an action:

```
// getter
$culture = $this->getUser()->getCulture();
// setter
$this->getUser()->setCulture('en_US');
```

This culture is persistent between pages because it is serialized in the user session.

Keeping both the language and the country in the culture is necessary because you may have a different French translation for users from France, Belgium or Canada, and a different Spanish translation for users from Spain or Mexico.

The language is coded in two lower-case characters, according to the ISO 639-1 norm (for instance `en` for English).

The country is coded in two upper-case characters, according to the ISO 3166-1 norm (for instance `GB` for Great-Britain).

By default, any new user will take the culture set in the `default_culture` configuration parameter. You can change it in the `i18n.yml` configuration file:

```
all:
  default_culture:     fr
```

All the culture-dependent contents are displayed transparently according to the user culture.

Standards and formats

Once the culture is defined, the helpers depending on it will automatically have a proper output. Here is a list of helpers that take into account the user culture for their output:

```
// formatting helpers
format_date($date, $format)
format_datetime($date, $format)
format_number($number)
format_currency($amount, $currency)
format_country($country_iso)
// form helpers
input_date_tag($name, $value, $options)
select_country_tag($name, $value, $options)
```

For instance,

```
<?php echo format_number(12000.10) ?>
// will generate in HTML with a culture set to en_US
12,000.10
// will generate in HTML with a culture set fr_FR
12 000,10
```

If you want to know more about the helpers that depend on culture, refer to the form helpers, i18n helpers and other helpers documentation.

Text information in the database

For each table that contains some i18n data, it is recommended to split the table in two parts: one table with no i18n column, and the other one with only the i18n columns. This setup lets you add more languages when needed without a change to your model. Let's take an example with a Product table.

First, create tables in the `schema.yml` file:

```
my_connection:
  my_product:
    _attributes: { phpName: Product, isI18N: true, i18nTable: my_product_i18n }
    id:          { type: integer, required: true, primaryKey: true, autoincrement: true }
    price:       { type: float }
  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    id:          { type: integer, required: true, primaryKey: true, foreignTable: my_product, for
    culture:     { isCulture: true, type: varchar, size: 7, required: true, primaryKey: true }
    name:        { type: varchar, size: 50 }
```

Notice the `isI18N` and `i18nTable` attributes of the first table key, and the special `culture` column. Also, the `_i18n` suffix of the second table is a convention that automates many data access mechanisms. All these are symfony specific Propel enhancements.

> **Note**: The symfony automatisms can make this much faster to write. If the table containing internationalized data has the same name as the main table with `_i18n` as a suffix, and that they are related with a cilumn named `id` in both tables, you can write the same as above with

only:

```
my_connection:
  my_product:
    _attributes: { phpName: Product}
    id:
    price:       float
  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    name:        varchar(50)
```

You can find more about the schema syntax and automatisms in the model chapter.

Once the corresponding object model is built (don't forget to call `symfony propel-build-model` and clear the cache with a `symfony cc` after each modification of the `schema.yml`), you can use your Product class with i18n support as if there was only one table:

```
$product = ProductPeer::retrieveByPk(1);
$product->setCulture('fr');
$product->setName('Nom du produit');
$product->save();

$product->setCulture('en');
$product->setName('Product name');
$product->save();

echo $product->getName();           =>    'Product name'
$product->setCulture('fr');
echo $product->getName();           =>    'Nom du produit'
```

Note: If you don't want to remember to change the culture each time you use an i18n object, you can also change the `hydrate` method in the object class. In the previous example, add the following function to the `myproject/lib/model/Product.php`:

```
public function hydrate(ResultSet $rs, $startcol = 1)
{
  parent::hydrate($rs, $startcol);
  $this->setCulture(SF_DEFAULT_CULTURE);
}
```

or even, to get the actual user culture:

```
public function hydrate(ResultSet $rs, $startcol = 1)
{
  parent::hydrate($rs, $startcol);
  $this->setCulture(sfContext::getInstance()->getUser()->getCulture());
}
```

Interface translation

Symfony stores the user interface translations in XML configuration files in the standard XLIFF format. Here is an extract of a XLIFF file `messages.fr.xml`, where a website originally written in English is translated in French:

```
<?xml version="1.0" ?>
<xliff version="1.0">
  <file orginal="global" source-language="en_US" datatype="plaintext" date="2004-12-28T18:10:19Z"
    <body>
      <trans-unit id="1">
        <source>original English text</source>
        <target>French translation of the text</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

If additional translations need to be done, simply add a new `messages.XX.xml` translation file. These files must be stored in the `apps/myapp/i18n` directory.

The use of the XLIFF format allows you to use common translation tools to reference all text in your website and translate it, without the need of a specific tool to build for translators.

Lets say you want to have a website in English and French, with English being the default language. The example page is supposed to display the phrase 'Welcome to our website. Today's date is ' followed by the actual date.

For this to work you need to:

1. Activate the i18N interface translation in the application `settings.yml`

   ```
   all:
     .settings:
       i18n: on
   ```
2. Activate the I18N helper by adding `,I18N` to `standard_helpers:` in the `settings.yml` file. Alternatively, you can put `<?php use_helper('I18N') ?>` at the top of the template where you want to use internationalisation.

   The same goes for the 'Date' helper to make sure the date is being formatted right.
3. In the template, enclose all the texts in calls to the `__()` function. So, to have an internationalized template displaying:

   ```
   Welcome to our website. Today's date is
   <?php echo format_date(date()) ?>
   ```

   It must be written like this:

   ```
   <?php echo __('Welcome to our website.') ?>     // (this matches the first trans-unit's so
   <?php echo __('Today's date is ') ?>            // (this matches the second trans-unit's s
   <?php echo format_date(date()) ?>
   ```
4. Make a `messages.fr.xml` file in the `app/i18n` directory, according to the XLIFF format, containing one `trans-unit` node for eah call to the `__()` function:

   ```
   <?xml version="1.0" ?>
   <xliff version="1.0">
     <file orginal="global" source-language="en_US" datatype="plaintext">
       <body>
   ```

```
            <trans-unit id="1">
              <source>Welcome to our website.</source>
              <target>Bienvenue sur notre site web</target>
            </trans-unit>
            <trans-unit id="2">
              <source>Today's date is </source>
              <target>La date d'aujourd'hui est </target>
            </trans-unit>
          </body>
        </file>
      </xliff>
```

Notice the `source-language="..."`: put the full iso code of your default culture here.

The default user culture is set to `en_US`, so the text outputs in English. However, if the culture is changed to `fr_BE` with:

```
$this->getUser()->setCulture('fr_BE');
```

...the text then outputs in French.

If, in the near future, you want to add a Dutch translation, all you have to do is duplicate `messages.fr.xml` and call it `messages.nl.xml`. Then, open the newly created file and change all the text contained in the `<target>` nodes to the dutch translation.

Note that translation only makes sense if the translation files contains full sentences. However, as you sometimes have formatting or variables in the text, you could be tempted to cut the sentence into several parts. For instance, to translate:

```
Welcome to all the <b>new</b> users.<br />
There are <?php echo count_logged() ?> persons logged.
```

You could to write:

```
<?php echo __('Welcome to all the') ?><b><?php echo __('new') ?> </b><?php echo __('users') ?> <b
<?php echo __('There are') ?><?php echo count_logged() ?><?php echo __('persons logged') ?>
```

But with this template, your translation file would be totally incomprehensible for translators. You'd better do:

```
<?php echo __('Welcome to all the <b>new</b> users') ?> <br />
<?php echo __('There are %1% persons logged', array('%1%' => count_logged())) ?>
```

The syntax of the last lines illustrates the ability to do a simple substitution within the `__()` function, to avoid unnecessary chunking of text.

One of the common problems with translation is the use of the plural form. According to the number of results, the text changes, such as in:

```
<?php echo __('There are %1% persons logged', array('%1%' => count_logged())) ?>
```

What if there is only one person logged? The message would be:

```
<?php echo __('There is 1 person logged') ?>
```

To avoid multiple tests, symfony provides a special translation helper called `format_number_choice()`. Use it like this:

```
<?php echo format_number_choice(
  '[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf] There are %1% persons logged',
  array('%1%' => count_logged()),
  count_logged()
) ?>
```

The first argument is the multiple possibilities of text. The second argument is the replacement pattern (as for the `___()` helper) and is optional. The third argument is the number on which the test is made to determine which text is taken.

The message/string choices are separated by the pipe "|" followed by a set notation of the form

- `[1,2]`: accepts values between 1 and 2, inclusive.
- `(1,2)`: accepts values between 1 and 2, excluding 1 and 2.
- `{1,2,3,4}`: only values defined in the set are accepted.
- `[-Inf,0)`: accepts value greater or equal to negative infinity and strictly less than 0

Any non-empty combinations of the delimiters of square and round brackets are acceptable.

How to build a syndication feed

Overview

Whether your application lists posts, images, news, questions or anything else, if the update rate is higher than once a month, you must provide a syndication feed (RSS, Atom, etc.) to your users so that they can keep up-to-date about your website from within a feed aggregator. The good news is, if your object model is built in the right way, this won't take you more than a couple lines to develop, since symfony provides a feed builder plugin that does it all for you.

Introduction

The example explored in this chapter is an simple blog application with a `Post` and an `Author` table:

Post	Author
id	id
author_id	first_name
title	last_name
description	email
body	
created_at	

The `Post` class is extended by a `->getStrippedTitle()` method that transforms the title into a string that can be used in an URI, replacing spaces by dashes, upper case by lower case, and removing all special characters:

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

  // strip all non word chars
  $text = preg_replace('/\W/', ' ', $text);
  // replace all white space sections with a dash
  $text = preg_replace('/\ +/', '-', $text);
  // trim dashes
  $text = preg_replace('/\-$/', '', $text);
  $text = preg_replace('/^\-/', '', $text);

  return $text;
}
```

The `Author` class is extended by a custom `->getName()` method as follows:

```
public function getName()
{
  return $this->getFirstName().' '.$this->getLastName()
}
```

If you need more details about the way to extend the model, refer to the model chapter.

The `routing.yml` contains the following rule:

```
post:
    url:   /permalink/:stripped_title
    param: { module: post, action: read }
```

If you need more details about the routing system, refer to the routing chapter.

A special `feed` module is built for the occasion, and all the actions and templates will be placed in it.

```
$ symfony init-module myapp feed
```

Expected result

The feed action has to output an Atom feed. As a reminder of all the information that need to be included in an Atom feed, here is an example:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

  <title>The mouse blog</title>
  <link href="http://www.myblog.com/" />
  <updated>2005-12-11T16:23:51Z</updated>
  <author>
    <name>Peter Clive</name>
    <author_email>pclive@myblog.com</author_email>
  </author>
  <id>4543D55FF756G734</id>

  <entry>
    <title>I love mice</title>
    <link href="http://www.myblog.com/permalink/i-love-mice" />
    <id>i-love-mice</id>
    <author>
      <name>Peter Clive</name>
      <author_email>pclive@myblog.com</author_email>
    </author>
    <updated>2005-12-11T16:23:51Z</updated>
    <summary>Ever since I bought my first mouse, I can't live without one.</summary>
  </entry>

  <entry>
    <title>A mouse is better than a fish</title>
    <link href="http://www.myblog.com/permalink/a-mouse-is-better-than-a-fish" />
    <id>a-mouse-is-better-than-a-fish</id>
    <author>
      <name>Bob Walter</name>
      <author_email>bwalter@myblog.com</author_email>
    </author>
    <updated>2005-12-09T09:11:42Z</updated>
    <summary>I had a fish for four years, and now I'm sick. They smell.</summary>
  </entry>

</feed>
```

Install the plug-in

Symfony provides a `sfFeed` plug-in that automates most of the feed generation. To install it, use the symfony command line:

```
$ symfony plugin-install local symfony/sfFeed
```

This installs the classes of the plug-in in the `myapp/lib/symfony/plugins/` directory, because the `local` option tells symfony to install the plug-in for the current application only. You could have installed it for all your projects by replacing `local` by `global`.

If you want to learn more about plug-ins, how they extend the framework and how you can package the features that you use across several projects into a plug-in, read the plug-in chapter of the symfony book.

Build the feed by hand

In the `feed` module, create a `lastPosts` action:

```
public function executeLastPosts()
{
  $feed = sfFeed::newInstance('atom1');

  $feed->setTitle('The mouse blog');
  $feed->setLink('http://www.myblog.com/');
  $feed->setAuthorEmail('pclive@myblog.com');
  $feed->setAuthorName('Peter Clive');

  $c = new Criteria;
  $c->addDescendingOrderByColumn(PostPeer::CREATED_AT);
  $c->setLimit(5);
  $posts = PostPeer::doSelect($c);

  foreach ($posts as $post)
  {
    $item = new sfFeedItem();
    $item->setFeedTitle($post->getTitle());
    $item->setFeedLink('@permalink?stripped_title='.$post->getStrippedTitle());
    $item->setFeedAuthorName($post->getAuthor()->getName());
    $item->setFeedAuthorEmail($post->getAuthor()->getEmail());
    $item->setFeedPubdate($post->getCreatedAt('U'));
    $item->setFeedUniqueId($post->getStrippedTitle());
    $item->setFeedDescription($post->getDescription());

    $feed->addItem($item);
  }

  $this->feed = $feed;
}
```

The initial factory method creates an instance of the `sfFeed` class for the 'Atom' format. The `sfFeed` and `sfFeedItem` classes are symfony add-ons made especially for feed construction. At the end of the action, the `$feed` variable contains a `sfFeed` object which includes several `sfFeedItem` objects. To transform the object into an actual Atom feed, the `lastPostsSuccess.php` template simply contains:

```
<?php echo $feed->getFeed() ?>
```

The template must not be decorated by a layout. In addition, the resulting page has to be declared as a `text/xml` content-type. So add a `view.yml` in the `feed` module `config/` directory:

```
all:
  has_layout: off

  http_metas:
    content-type: text/xml
```

When called from a feed aggregator, the result of the action is now exactly the Atom feed described above:

```
http://www.myblog.com/feed/lastPosts
```

Use the short syntax

The use of all the setters for the item construction can be a little annoying, since there is a lot of information to define. Symfony provides a short hand syntax that produces the same effect, using an associative array:

```
public function executeLastPosts()
{

  $feed = sfFeed::newInstance('atom1');

  $feed->setTitle('The mouse blog');
  $feed->setLink('http://www.myblog.com/');
  $feed->setAuthorEmail('pclive@myblog.com');
  $feed->setAuthorName('Peter Clive');

  $c = new Criteria;
  $c->addDescendingOrderByColumn(PostPeer::CREATED_AT);
  $c->setLimit(5);
  $posts = PostPeer::doSelect($c);

  foreach ($posts as $post)
  {
    $item = array(
      'title'       => $post->getTitle(),
      'link'        => '@permalink?stripped_title='.$post->getStrippedTitle(),
      'authorName'  => $post->getAuthor()->getName(),
      'authorEmail' => $post->getAuthor()->getEmail(),
      'pubdate'     => $post->getCreatedAt(),
      'uniqueId'    => $post->getStrippedTitle(),
      'description' => $post->getDescription(),
    );

    $feed->addItemFromArray($item);
  }

  $this->feed = $feed;
}
```

It has exactly the same effect, but the syntax is clearer.

Let symfony do it for you

As the method names that are used to build a news field are more or less always the same, symfony can do exactly as above with only:

```
public function executeLastPosts()
{

  $feed = sfFeed::newInstance('atom1');

  $feed->setTitle('The mouse blog');
  $feed->setLink('http://www.myblog.com/');
  $feed->setAuthorEmail('pclive@myblog.com');
  $feed->setAuthorName('Peter Clive');

  $c = new Criteria;
  $c->addDescendingOrderByColumn(PostPeer::CREATED_AT);
  $c->setLimit(5);
  $posts = PostPeer::doSelect($c);

  $feed->setFeedItemsRouteName('@permalink');
  $feed->setItems($posts);

  $this->feed = $feed;
}
```

Isn't it pure magic?

The magic of the `sfFeed` object

The getter functions of the `Post` object are so clear that even symfony can understand them. That's because the `sfFeed` class has built-in mechanisms to deduce relevant information from well-organised classes:

- To set the item `title`, it looks for a `->getFeedTitle()`, a `->getTitle()`, a `->getName()` or a `->__toString()` method.

  In the example, the `Post` object has a `->getName()` method.
- To set the `link`, it looks for an feed item route name (defined by a `setFeedItemsRouteName()` call). If there is one, it looks in the route url for parameters for which it could find a getter in the object methods. If not, it looks for a `->getFeedLink()`, `->getLink()`, `->getUrl()` method.

  In the example, one route name is defined above in the action (`@permalink`). The routing rule contains a `:stripped_title` parameter and the `Post` object has a `->getStripped_Title()` method, so the `sfFeed` object is able to define the URIs to link to.
- To set the author's email, it looks for a `getFeedAuthorEmail` or a `getAuthorEmail`. If there is no such method, it looks for a `->getAuthor()`, `->getUser()` or `->getPerson()` method. If the result returned is an object, it looks in this object for a `getEmail` or a `getMail` method.

  In the example, the `Post` object has a `->getAuthor()`, and the `Author` object has a `->getName()`. The same kind of rules is used for the author's name and URL.

- To set the publication date, it looks for a `->getFeedPubdate()`, `->getPubdate()`, `->getCreatedAt()` or a `->getDate()` method.

  In the example, the `Post` object has a `getCreatedAt`

The same goes for the other possible fields of an Atom feed (including the categories, the summary, the unique id, etc.), and you are advised to browse the source of the `sfFeed` class to discover all the deduction mechanisms.

All in all, the way the accessors of the `Post` and `Author` objects are built allow the built-in shortcuts of the `sfFeed` to work, and the creation of the feed to be so simple.

Define custom values for the feed

In the list presented above, you can see that the first method name that the `sfFeed` object looks for is always a `->getFeedXXX()`. This allows you to specify a custom value for each of the fields of a feed item by simply extended the model.

For instance, if you don't want the author's email to be published in the feed, just add the following `->getFeedAuthorEmail()` method to the `Post` object:

```
public function getFeedAuthorEmail()
{
  return '';
}
```

This method will be found before the `->getAuthor()` method, and the feed will not disclose the publishers' email addresses.

Use other formats

The methods described below can be transposed to build other RSS feeds. Simply change the parameter given to the feed factory:

```
// RSS 0.91
$feed = sfFeed::newInstance('rssUserland091');
// RSS 2.01
$feed = sfFeed::newInstance('rss201rev2');
```

How to paginate a list

Overview

Symfony provides a pager component: the `sfPropelPager` object. It can separate a list of results from a criteria object (of `Criteria` class) into a set of pages for display, and offers access methods to pages and result objects.

The `sfPropelPager` object

The `sfPropelPager` class uses the Propel abstraction layer, as described in the Model chapter.

This chapter will illustrate the way to use the `sfPropelPager` methods with a simple example : displaying a list of articles ten by ten. Assume that the `Article` object has `->getPublished()`, `->getTitle()`, `->getOverview()` and `->getContent()` accessor methods.

If you wanted to have the non-paginated result of a criteria request showing only published articles, you would need:

```
class articleActions extends sfActions
{
  public function executeList()
  {
    ...
    $c = new Criteria();
    $c->add(ArticlePeer::PUBLISHED, true);
    $articles = ArticlePeer::doSelect($c);
    $this->articles = $articles;
    ...
  }
}
```

The `$articles` variable, accessible from the template, would contain an array of all the `Article` objects matching the request.

To get a paginated list, you need a slightly different approach; the results have to be put in a `sfPropelPager` object instead of an array:

```
class articleActions extends sfActions
{
  public function executeList()
  {
    ...
    $c = new Criteria();
    $c->add(ArticlePeer::PUBLISHED, true);
    $pager = new sfPropelPager('Article', 10);
    $pager->setCriteria($c);
    $pager->setPage($this->getRequestParameter('page', 1));
    $pager->init();
    $this->pager = $pager;
    ...
```

```
  }
}
```

The differences start after the criteria definition, since this action:

- creates a new pager to paginate `Article` objects ten by ten
- affects the criteria to the pager
- sets the current page to the requested page or to the first one
- initializes the pager (i.e. execute the request related to the criteria)
- passes the pager to the template via the `$pager` variable

The template `listSuccess.php` can now access the `sfPropelPager` object. This object knows the current page and the list of all the pages. It has methods to access pages and objects in pages. Let's see how to manipulate it.

To display the total number of results, use the `->getNbResults()` method:

```
<?php echo $pager->getNbResults() ?> results found.<br />
Displaying results <?php echo $pager->getFirstIndice() ?> to  <?php echo $pager->getLastIndice()
```

To display the articles of the requested page, use the `->getResults()` method of the `pager` object to retrieve the objects in the page:

```
<?php foreach ($pager->getResults() as $article): ?>
  <?php echo link_to($article->getTitle(), 'article/read?id='.$article->getId()) ?>
  <?php echo $article->getOverview() ?>
<?php endforeach ?>
```

Navigating across pages

The pager object knows if the number of results exceeds the maximum number that can be displayed in one page (10 in this example) thanks to the `->haveToPaginate()` method.

To add the page navigation links at the bottom of the list (Â« < > Â»), use the navigation methods `->getFirstPage()`, `->getPreviousPage()`, `->getNextPage()` and `->getLastPage()`. The current page is given by `->getPage()`. All these methods return an integer : the rank of the requested page.

To point to a specific page, loop through the collection of links obtained by a call to the `->getLinks()` method:

```
<?php if ($pager->haveToPaginate()): ?>
  <?php echo link_to('&laquo;', 'article/list?page='.$pager->getFirstPage()) ?>
  <?php echo link_to('&lt;', 'article/list?page='.$pager->getPreviousPage()) ?>
  <?php $links = $pager->getLinks(); foreach ($links as $page): ?>
    <?php echo ($page == $pager->getPage()) ? $page : link_to($page, 'article/list?page='.$page)
    <?php if ($page != $pager->getCurrentMaxLink()): ?> - <?php endif ?>
  <?php endforeach ?>
  <?php echo link_to('&gt;', 'article/list?page='.$pager->getNextPage()) ?>
  <?php echo link_to('&raquo;', 'article/list?page='.$pager->getLastPage()) ?>
<?php endif ?>
```

This should render something like:

« < 1 - 2 - 3 - 4 - 5 > »

Once the article displayed, to allow a direct navigation to the previous or the next article without going back to the paginated list, you will need a cursor.

Navigating across objects

Navigating page by page within the list is easy, but the users might not want to go back to the list to navigate object by object. The `cursor` attribute of the `sfPropelPager` object can hold the offset of the current object.

This will allow an article by article navigation in the `readSuccess.php` template. First, let's modify a bit of the code of the `listSuccess.php` template:

```
<?php $cursor = $pager->getFirstIndice(); foreach ($pager->getResults() as $article): ?>
  <?php echo link_to($article->getTitle(), 'article/read?cursor='.$cursor) ?>
  <?php echo $article->getOverview() ?>
<?php ++$cursor; endforeach ?>
```

The `read` action will need to know how to handle a `cursor` parameter:

```
class articleActions extends sfActions
{
  public function executeRead()
  {
    ...
    if ($this->getRequestParameter('cursor'))
    {
      $article = $pager->getObjectByCursor($this->getRequestParameter('cursor'));
    }
    else if ($this->getRequestParameter('id'))
    {
      $article = ArticlePeer::retrieveByPK($this->getRequestParameter('id'));
    }

    // Error
    $this->forward404Unless($article);
  }
}
```

The `->getObjectByCursor($cursor)` method sets the cursor at a specified position and returns the object at that very position.

You can set the cursor without getting the resulting object with the `->setCursor($cursor)` method. And once the cursor is set, you can grab the current object at this position (`->getCurrent()`) but also the previous one (`->getPrevious()`) and the next one (`->getNext()`).

This means that the `read` action can pass to the template the necessary information for an article-by-article navigation with a few modifications:

```
class articleActions extends sfActions
{
  public function executeRead()
  {
    ...
    if ($this->getRequestParameter('cursor'))
    {
      $pager->setCursor($this->getRequestParameter('cursor'));
      $previous_article = $pager->getPrevious();
      $article = $pager->getCurrent();
      $next_article = $pager->getNext();
    }
    else if ($this->getRequestParameter('id'))
    {
      $article = ArticlePeer::retrieveByPK($this->getRequestParameter('id'));
    }

    // Error
    $this->forward404Unless($article);
  }
}
```

> **Note**:The `->getPrevious()` and `->getNext()` methods return `null` if there is no previous or next object.

The `readSuccess.php` template could look like:

```
<h1><?php echo $article->getTitle() ?></h1>
<p class="overview"><?php echo $article->getOverview() ?></p>
<div class="content">
  <?php echo $article->getContent() ?>
</div>
&lt; <?php echo link_to_if($previous_article, $previous_article->getTitle(), 'article/read?id='.$
-
&gt; <?php echo link_to_if($next_article, $next_article->getTitle(), 'article/read?id='.$next_art
```

Changing the sort order

As the `sfPropelPager` object relies on a `Criteria` object, changing the order of the pager is simply done by adding a sort to the criteria, before assigning it to the pager object.

For instance, you can add the choice of the sort column to the list navigation interface:

```
class articleActions extends sfActions
{
  public function executeList()
  {
    ...
    $c = new Criteria();
    $c->add(ArticlePeer::PUBLISHED, true);
    if ($this->getRequestParameter('sort'))
    {
      $c->addAscendingOrderByColumn(ArticlePeer::translateFieldName($this->getRequestParameter('s
    }
    else
    {
```

```
      // sorted by date by default
      $c->addAscendingOrderByColumn(ArticlePeer::UPDATED_AT);
    }
    $pager = new sfPropelPager('Article', 10);
    $pager->setCriteria($c);
    $pager->init();
    $this->pager = $pager;
    ...
  }
}
```

Add the following to the `listSuccess.php` template:

```
Sort by : <?php echo link_to('Title', 'article/list?sort=title') ?> - <?php echo link_to('Id', 'a
```

Changing the number of results per page

The `->setMaxPerPage($max)` method changes the number of results displayed per page, without the need to reprocess the pager (no need to call `init()` again). If you pass the value `0` for parameter, the pager will display all the results in one single page.

```
class articleActions extends sfActions
{
  public function executeList()
  {
    ...
    $c = new Criteria();
    $c->add(ArticlePeer::PUBLISHED, true);
    $pager = new sfPropelPager('Article', 10);
    $pager->setCriteria($c);
    if ($this->getRequestParameter('maxperpage'))
    {
      $pager->setMaxPerPage($this->getRequestParameter('maxperpage'));
    }
    $pager->init();
    $this->pager = $pager;
    ...
  }
}
```

So you can add the following to the `listSuccess.php` template:

```
Display : <?php echo link_to('10', 'article/list?maxperpage=10' ?> - <?php echo link_to('20', 'ar
```

Changing the select method

If you need to optimize the performance of an action relying on a `sfPropelPager`, you might want to force the pager to use a `doSelectJoinXXX()` instead of a simple `doSelect()`. This is easily achieved by the `->setPeerMethod()` method of the `sfPropelPager` object:

```
$pager->setPeerMethod('doSelectJoinUser');
```

Note that the pager actually processes the `doSelect()` query when displaying a page. The first query (triggered by `$pager->init()`) does only a `doCount`, and you can also customize this method by calling:

```
$pager->setPeerCountMethod('doCountJoinUser');
```

Storing additional information in the pager

You may need sometimes to keep a certain context in a pager object. That's why the `sfPropelPager` class can handle *parameters* in the usual way:

```
$pager->setParameter('foo', 'bar');

if ($pager->hasParameter('foo'))
{
  $pager->getParameter('foo');
  $pager->getParameterHolder()->removeParameter('foo');
}

$pager->getParameterHolder()->clearParameters();
```

These parameters are never used directly by the pager.

To learn more about custom parameters, refer to the parameter holder chapter.

How to generate a Propel scaffolding or administration

Overview

Many applications are based on data stored in a database, and offer an interface to access it. Symfony automates the repetitive task of creating an initial CRUD or a full featured backend administration based on a Propel object with a few handy generators.

Data manipulation: The needs

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

- Create a record
- Retrieve records (requesting the database of one or more records)
- Update a record (and modify its fields)
- Delete a record

These operations are so common that they have a dedicated acronym: C.R.U.D.

Many pages can be reduced to one of these actions. For instance, in a forum application, the list of latest posts is a Retrieve action, and the reply to a post is a Create action. During the process of application creation, developers often need to redevelop the basic actions and template that implement the CRUD operations for a given table. And even if there are many ways to code it, the Propel layer offers consistant setters and getters, so it can be automated for the most part.

This brings up two interesting automated mechanisms based on a table: The generation of a scaffolding and the generation of an administration. They are not used for the same purpose, and symfony provides different utilities for these two, so it is best if you understand clearly their differences:

- A **scaffolding** is the basic structure (actions and templates) required to operate CRUD on a given table. The code is often minimal, and the formatting is neglected, since the purpose of a scaffolding is to serve as a guideline for a much more ambitious construction. For instance, when you start building an application that will handle posts (like a forum), you should start by generating a scaffolding based on the `Post` table. The scaffolding will give you the CRUD actions and the related templates, all that working together, and ready to be modified, combined, or erased, according to your needs. Beware though that scaffolding doesn't provide pagination, validation or fine configuration possibilities. It is a quick tool to give you a starting base.
- 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. Their presentation is important, and they take advantage of additional features such as sorting, pagination and filtering.

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

Initiating or generating

Symfony offers two ways to create CRUD interfaces: Either by inheritance or by code generation.

- You can create empty classes that **inherit** from the framework. This masks the php code of the actions and the templates to avoid them being modified. This is useful if your data structure is not final, or if you want to completely rewrite some of the actions and keep the others. It also allows you to take advantage of future upgrades in the generators, because the code executed at runtime is not located in your application, but in the framework. The command line tasks for this kind of generation start with:

  ```
  $ symfony propel-init-
  ```

  It will create empty actions like the following:

  ```
  <?php

          class articleActions extends autoarticleActions
          {
          }

          ?>
  ```

- You can also **generate** the code of the actions and the templates so that it can be modified. The resulting module is independant from the classes of the framework, and cannot be altered using configuration files (see below). The command line tasks for this kind of generation start with:

  ```
  $ symfony 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 keep usable even if the data model changes. That's why administrations are often initiated only.

Scaffolding

Generating a scaffolding

When you need a basic set of usable actions and templates that can access and modify the data of a given table, generate a scaffolding. For instance, to generate the scaffolding for an `article` module based on the `Article` model class, type:

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

This will create a new `modules/article/` directory in your `myapp` application, with the following code:

Action	Purpose
`index`	forwards to the list action below
`list`	displays the list of all the records of the table
`show`	

	displays the detailed view (line by line) of one record
`edit`	displays a form to modify the fields of a record
`update`	action called by the form of the edit action above
`delete`	deletes a record
`create`	creates a new record

Template	Purpose
`editSuccess.php`	record edition form
`listSuccess.php`	list of all records
`showSuccess.php`	detail of one record

The module can be used as soon as it is generated:

```
http://myapp.example.com/index.php/article
```

Using this module, you can create new articles, modify or delete existing ones. The generated code is ready to be modified for your own needs. As an example, here is the beginning of the `actions.class.php` file:

```
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 instanceof Article);
  }
  ...
```

Repeat the CRUD generation for all the tables that you want to interact with, and you have a working scaffolding for a whole web application.

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:

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

Access it with the default action:

```
http://myapp.example.com/index.php/article
```

And start using your simple application just the way you would with a generated one.

If you check the newly created `actions.class.php` in the `article` module, you will see that it is empty: Everything is inherited from a Propel CRUD generator 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 framework. It can be found in the application cache (`myproject\cache\myapp\prod\module\autoArticle\`).

Administration

Symfony can create modules based on propel objects for the backend of your applications. Contrary to other generators, you keep the total control of the modules, since you benefit from the usual module mechanisms (decorator, validation, routing, custom configuration, autoloading, etc.). You are free to link the modules as you want, or to add a module of your own. You can also override any piece of action or template.

Setting up an administration is as easy as initiating a scaffolding:

```
$ symfony propel-init-admin myapp article Article
```

Just like scaffoldings, admin modules are related to a Propel object (`Article` in the example above). The actions created are the almost the same (except there is no `show` action), and can be accessed the same way:

```
http://myapp.example.com/index.php/article
```

The main difference is that an admin relies on a configuration file called `generator.yml`. To see the default configuration of an administration module just created, open the `myapp/modules/article/config/generator.yml` file:

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

To customize the generator, you don't need to modify the `article` actions, but rather change this configuration. You will learn more about the way to customize a generated administration module in the admin generator chapter.

The admin generator

Overview

Backend administrations are often built according to the structure of the data used in the frontend application. Such backends can be entirely generated by symfony, provided your object model is well defined. The mechanism that does that is called the **admin generator**. It automates many repetitive tasks using a system of custom configuration. It also gives you total control, allowing you to customize or extend all the generated components and the look and feel of the application. And when the admin generator can't fulfill your requirements, it provides the tools to plug your own code into the generated administration.

> **Note**: A screencast showing an administration being built is available for download (21min, QT7, 17 Mb).
>
> **Note**: A cheat sheet with the admin generator parameters explained briefly in a single page is available for download (pdf, 28 Kb).
>
> **Note**: If you need a basic CRUD interface to initiate the templates and actions of a module for your frontend application, you will probably prefer making a scaffolding than a generated admin.

Introduction

For this chapter, the example used will be a blog application with two `Article` and `Comment` classes, based the following table structure:

blog_article	blog_comment
id	id
title	article_id
content	author
	date
	content

The application in which the administration will be built is called `backend`.

Initiating an admin module

With symfony, you build an administration module by module. A module is generated based on a Propel object, using the command line interface with the `propel-init-admin` task, with `<APPLICATION_NAME>`, `<MODULE_NAME>` and `<CLASS_NAME>` as parameters:

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

This single command is enough to create an `article` module with `list`, `edit`, `create` and `delete` actions, based on the `Article` Propel class, and accessible by:

```
http://www.example.com/backend.php/article
```

The look and feel of a generated module is sophisticated enough to make it usable out of the box for a commercial application.

Modules generated by the admin generator benefit from the usual module mechanisms (decorator, validation, routing, custom configuration, autoloading, etc.). You are free to link the modules as you want, or to add a module of your own.

The generated code

If you look at the generated code in the `apps/backend/modules/article/` directory, you will find an empty action class and no templates. This is because the module inherits from classes and files located in the framework file structure. You can find the code which is actually used in the `cache/backend/ENV/modules/article` folder, once you've used it:

```
actions/actions.class.php
  create -> edit
  delete
  edit
  index -> list
  list
  save -> edit

templates/
  _edit_actions.php
  _edit_footer.php
  _edit_header.php
  _filters.php
  _list_actions.php
  _list_footer.php
  _list_header.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 admin 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.

Inheriting actions and templates allows you to override them completely. For instance, you can use your filters by adding your own `_filters.php` in the `modules/articles/templates/` directory.

The `generator.yml` configuration file

Most of the customization of a generated administration is done through a YAML configuration file called `generator.yml`. To see the default configuration of an administration module just created, open the `backend/modules/article/config/generator.yml` file:

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

This configuration is enough to generate the basic administration. The customization is added under the `param:` key (which means that all lines added in the `generator.yml` mus at least start with `four` blank spaces). Here is a typical customized `generator.yml` for a `post` module (extracted from the screencast source):

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

    fields:
      author_id:      { name: Post author }

    list:
      title:          symfony blog posts
      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 %%cat
      filters:        [title, category_id, author_id, is_published]
      max_per_page:   2

    edit:
      title:          Editing post "%%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 a blog author << }
        published_on: { credentials: [[admin, superdamin]] }
        content:      { params: rich=true tinymce_options=height:150 }
```

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

Fields

Fields settings

The configuration file can define how columns appear in the pages. For instance, to define a custom label for the `title` and `content` columns in the `article` module, edit the `generator.yml`:

```
generator:
  class:              sfPropelAdminGenerator
  param:
```

```
    model_class:      Article
    theme:            default

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

In addition to this default definition for all the views, you can override the `fields` settings for a given view (`list` and `edit`):

```
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: The settings that are set for the whole module under the `param` key can be overridden for a given view.

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, edit the `modules/comment/config/generator.yml`:

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

    fields:
      id:             { name: Id }
      article_id:     { name: Article }
      author:         { name: Author }
      date:           { name: Published on }
      content:        { name: Body }

    list:
      display:        [id, article_id, date]

    edit:
      display:
        "NONE":       [id, article_id]
```

```
        "Editable":   [date, author, content]
```

If you don't supply any group name (like in the `list` view above), put the fields that you want to display in an ordered array. If you want to group fields, use an associative array with the group name as a key, or `"NONE"` for a group with no name.

Custom fields

As a matter of fact, the `fields` parameters don't need to be actual columns in the tables. If you define a **custom getter and setter**, it can be used as a field as well. For instance, if you extend the `Article.class.php` model by a `->getNbComments()` method:

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

You can use `nb_comments` as a field in the admin (notice that the getter uses a camelCase version of the field name):

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

    fields:
      title:          { name: Article Title }
      content:        { name: Body }
      nb_comments:    { name: Number of comments }

    list:
      display:        [id, title, nb_comments]

    ...
```

Custom fields can even return HTML code to display more than raw data. For instance, if you extend the `Comment` class in the model with a `->getArticleLink()` method:

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

You can use it in the `comment/list` view by editing the related `generator.yml`:

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

    fields:
      id:             { name: Id }
```

```
      article_link:    { name: Article }
      author:          { name: Author }
      date:            { name: Published on }
      content:         { name: Body }

    list:
      display:         [id, article_link, date]
      ...
```

Partial fields

Beware that the code located in the model must be independent from the presentation. The example of the `getArticleLink()` method presented above doesn't respect this principle of layer separation. To achieve the same goal in a conceptually correct way, you'd better put the code that outputs HTML for a custom field in a partial. Fortunately the admin generator allows you to do it if you declare a field with a name starting by _ (and, in that case, you don't need to add custom methods to the model):

```
...
 list:
   display:         [id, _article_link, date]
```

For this to work, just add the following `_article_link.php` partial in the `modules/comment/templates/` directory:

```
<?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.

> **Note**: If you need to customize the parameters of a partial field (`_article_link` in this example), do the same as for a normal field, under the `field:` key. Just don't include the first underscore (_):
>
> ```
> fields:
>    article_link:    { name: Article }
> ```

If your partial gets crowded with more and more logic, you'll probably want to replace it by a component. Change the _ prefix by a ~ and you can define a component column:

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

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

Custom and partial fields can be used in the list view and in the edit view.

Customizing the views

Title

In addition to a custom set of fields, the `list` and `edit` pages can have a custom page title. In the string values of the `generator.yml`, the value of a field can be accessed via the name of the field surrounded by `%%`. For instance, if you want to customize the `article` views:

```
list:
   title:          List of Articles
   display:        [title, content, nb_comments]

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

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:

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

In the `list` view, the tooltip will be displayed in the column header, and in the `edit` view, the tooltip will be displayed close to each field.

Date format

Dates can be displayed using a custom format as soon as you use the `date_format` param:

```
list:
   fileds:
     date:         { name: Published, params: date_format='dd/MM' }
```

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

`list` view specific customization

Layout

The default list layout is the `tabular` layout, but you can also use the `stacked` layout. A field name preceded by = will contain a hyperlink to the detail of the related record. For instance, if you want to customize the `article/list` views:

```
list:
```

```
    title:          List of Articles
    layout:         tabular
    display:        [=title, content, nb_comments]
```

And the `comment/list` view:

```
  list:
    title:          List of Comments
    layout:         stacked
    params:         %%=content%% (sent by %%author%% on %%date%% about %%article_link%%)
    display:        [date, author, content]
```

Notice that a `tabular` layout expects a `display`, but a `stacked` layout uses the `params` key for the HTML code generated for each record. However, the `display` param is stille used in a `stacked` layout which columns are available for the interactive sorting.

Filters

In a `list` view, you can add a filter interaction, to help the user find a given set of records. For instance, to add filters to the `comment/list` view:

```
  list:
    filters:        [author, article_id]
```

The resulting filter will allow text-based search on an author name (where the `*` character can be used as a joker), and the selection of the comments related to a given article by a choice in a selection list. As for regular `object_select_tag()`, the options displayed in a select are the ones returned by the `->toString()` method of the related class (or the primary key if such a method doesn't exist).

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 only contain 2 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 string type) would be a text-based search, but what you want is probably a select with a list of values. That's quite easy to achieve with a partial filter:

```
  list:
    filters:        [date, _state]
```

Then, in `templates/_state.php`:

```
<?php echo select_tag('filters[state]', options_for_select(array(
  '' => '',
  'open' => 'open',
  'closed' => 'closed',
), isset($filters['state']) ? $filters['state'] : '')) ?>
```

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

Sort

In a `list` view, the column names are hyperlinks that can be used to reorder the list. These names are displayed both in the `tabular` and `stacked` layouts. Only the fields that correspond to an actual column are clickable - not the ones for custom or partial columns.

Clicking on these links reloads the page with a `sort` parameter. 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=date&type=desc' ) ?>
```

You can also define a default sort field for the `list` view directly in the `generator.yml`:

```
list:
   sort:   date
```

Or, if you want to specify the sort order:

```
list:
   sort:   [date, desc]
```

Pagination

The generated administration deals with large tables like a charm. The `list` view uses pagination by default, and you can customize the number of records to be displayed in each page with the `max_per_page` parameter:

```
list:
   max_per_page:   5
   title:          List of Comments
   layout:         stacked
   params:         %%=content%% (sent by %%author%% on %%date%% about %%article_link%%)
```

Join

The default method used by the admin generator to get the list is a `doSelect()`. But, if you use related objects in the list, the number database queries executed will rapidly increase. For instance, if you want to display the name of the author in a list of posts, an additional query will be made to the database for each line in the list to retrieve the related `Author` object.

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:   doSelectJoinAuthor
```

`edit` view specific customization

Input type

In an `edit` view, the user can modify the value of each field. Symfony determines the type of input to be used according to the data type of the column. For instance, fields finishing wirh `_id` will be displayed as select inputs. However, you may want to force a certain type of input for a given field, or the options of the object_tag generated. This kind of parameter goes into the `fields` definition:

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

    fields:
      id:             { name: Id }
      article_id:     { name: Article }
      author:         { name: Author }
      date:           { name: Published on }
      content:        { name: Body }

    ...

    edit:
      fields:
        id:           { type: plain }                              ## Drop the input, just displa
        author:       { params: disabled=true }                    ## The input is not editable
        content:      { type: textarea_tag, params: rich=true css=user.css tinymce_options=width:
        article_id:   { params: include_custom=Choose an article } ## The input is a select (obje

      ...
```

The `params` parameters are passed as options to the generated object_tag. For instance, the `params` definition for the `article_id` above will produce:

```
<?php echo object_select_tag($comment, 'getArticleId', 'related_class=Article', 'include_custom=C
```

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

Partial fields handling

Partial fields can be used in `edit` views just like in `list` views. The difference is that you have to plug the control(s) of the partial with the fields of the objects, by hand, in the action. For instance, an administration module for a `User` model object where the available fields are `id`, `nickname` and `password`, will probably not display the `password` field in clear, for security reasons. Instead, such modules often offer an empty `password` input that the user has to fill to change the value:

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

The `templates/newpassword.php` partial contain something like:

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

Notice that this partial uses a simple form helper, not an object form helper, since it is not intended to get the value from the current object.

Now, in order to use the value from this control to update the object in the action, you need to extend the `updateUserFromRequest()` method. To do that, create the following function in the module `actions.class.php` with the custom behaviour for the input of the partial field:

```
class autoUserActions extends sfActions
{
  protected function updateUserFromRequest()
  {
    $password = $this->getRequestParameter('newpassword');

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

    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, this is done via a validator. The admin generated modules benefit from this mechanism just like regular modules.

Table relationships

One to many

The 1-n table relationships are taken care of by the admin generator. In the example mentionned above, the `weblog_comment` table is related to the `weblog_article` table via the `article_id` field. If you initiate the module of the `Comment` class with the admin generator:

```
$ symfony propel-init-admin myapp comment Comment
```

The `comment/edit` action will automatically display the `article_id` as a select list showing the ids of the available records of the `weblog_article` table. In addition, if you define a `__toString()` method in the `Article` object, the string returned by this method is used instead of the ids in the select list.

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 the way of a partial field.

Many to many

The n-n table relationships are also taken care of. The implementation of such relationships is made through an intermediate table. For instance, if there is a 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), then your database will always end up with a table called `blog_article_author` or similar.

blog_article	blog_article_author	blog_author
id	article_id	id
title	author_id	name
...		...

The model will then have a class called `ArticleAuthor` and this is the only thing that the admin generator needs - but you have to pass it as a parameter of the field, called `trough_class`.

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.

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

Such a field handles links between existing objects, so a regular select 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. You have the choice between an `admin_double_list`, an `admin_select_list` and an `admin_check_list`.

Interactions

Admin modules are made to interact with the data. The basic interactions that can be performed are the usual CRUD, but you can also add your own interactions or restrict the possible interactions for a view. For instance, the following interaction definition for the `article` module gives access to all the CRUD actions:

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

 edit:
   title:          Body of article %%title%%
   actions:
     _list:        -
     _save:        -
     _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. In an `edit` view, as there is only one record edited at a time, there is only one set of actions to define.

The `_XXX` lines tell symfony to use the default icon and action for these interactions. But you can also add a custom interaction:

```
list:
```

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

You also have to define the `addForArticle` in `actions.class.php`:

```
public function executeAddComment()
{
  $comment = new Comment();

  $comment->setArticleId($this->getRequestParameter('id'));

  $comment->save();

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

Notice that symfony is smart enough to pass the primary key of the object for which the action was called as a request parameter.

One last word about actions: If you want to suppress completely the actions for one category, use an empty list:

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

Form validation

If you take a look at the generated `editSuccess.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 are made with the name of the module concatenated to the name of the field between angle brackets.

For instance, if the `edit` view for the `article` module has a `title` field, the template will look like:

```
<?php echo object_input_tag($article, 'getTitle', array('control_name' => 'post[title]')) ?>
// will generate in html
<input type="text" name="article[title]" id="article[title]" value="My Title" />
```

This has plenty of advantages during the form handling process. However, it makes the form validation configuration a bit trickier, since this format messes up with the YAML syntax. For instance, you can't write in a validation file for an admin `edit` view:

```
methods:
  names: [article[title], article[body]]
```

You will have to enclose the field names between double quotes `" "` and change square brackets `[ ]` by curly brackets `{ }`. Additionally, to avoid YAML interpretation problems, you should use the explicit YAML syntax for arrays. It makes the validation file look like:

```
methods:
  post:
    - "article{title}"
    - "article{body}"
```

When declaring the validators for a field, you don't need the double quotes but you must keep the curly brackets:

```
names:
  article{title}:
    required:     Yes
    required_msg: You must provide a title
```

Lastly, when using a field name as a parameter for a validator, you should use the name as it appears in the generated HTML code, i.e. with the square brackets, but between quotes:

```
passwordValidator:
  class:          sfCompareValidator
  param:
    check:        "user[newpassword]"
    compare_error: The password confirmation does not match the password. Please try again.
```

Presentation

Custom stylesheet

You can also define an alternative CSS to be used for an admin module instead of a default one:

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

... or override the styles per view, using the usual mechanisms provided by the module `view.yml` configuration. Since the generated HTML is structured content, you can do pretty much anything you like with the presentation.

Custom header and footer

The `list` and `edit` views can include a custom `header` and `footer` partial. There is no such partial by default in the `templates/` directory of an admin 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` in the `modules/articles/template/` directory with the following content:

```
<?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 list of objects through the plural form variable (`$articles` for this module).

Custom template parts

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

```
_edit_actions.php
_filters.php
_list_actions.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
```

Get the default version from the `symfony/data/generator/sfPropelAdmin/default/template/templates/` directory.

Calling the admin actions with custom parameters

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

```
<?php if($article->getNbComments()>0): ?>
  <?php echo link_to('View the '.$article->getNbComments().' comments to this article.', 'comment
<?php endif; ?>
```

The query string presented above is an encoded version of the more legible:

```
'filter=filter                              # filters are to be reseted with the following params
 filters[article_id]='.$article->getId(); # filter the comments to display only the ones related
```

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.

Credentials

For a given admin module, the elements displayed and the actions available can vary according to the credentials of the logged user.

The generator can take a `credentials` parameter into account to hide an element to users who don't have the proper credential if you use it in the `fields` section:

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

This works for the `list` view and the `edit` view.

The generator can also hide interactions according to credentials:

```
## 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: back
```

The `credentials` parameter accepts the usual credentials syntax, which allows you to combine credentials with `AND` and `OR`:

```
credentials: [ admin, superuser ]           ## admin AND superuser
credentials: [[ admin, superuser ]]         ## admin OR superuser
credentials: [[ admin, superuser ], owner] ## (admin OR superuser) AND owner
```

If you want to learn more about credentials, please refer to the security chapter of the symfony book.

Customize the theme

If you customize several modules in the same way, you should probably create a **theme** that could be reused across modules. The `theme` defined at the beginning of the `generator.yml` can be changed to use an alternative set of templates and stylesheets. With the `default` theme, symfony uses the files defined in `symfony/data/generator/sfPropelAdmin/default`, and you can create a new theme in the framework to override any of the actions or templates.

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

To create your own theme, add a new directory in the `symfony/data/generator/sfPropelAdmin/` folder and fill it with your custom version of the following elements, using the same structure as the `default/` theme:

```
fragments:
_edit_actions.php
_edit_footer.php
_edit_header.php
_filters.php
```

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

actions:
processFilters()     // process the request filters
addFiltersCriteria() // adds a filter to the Criteria object
processSort()
addSortCriteria()
```

Translation

All the texts that are in the generated templates are already internationalized (i.e. enclosed in a call to the `__('')` function). This means that you can easily translate a generated admin by adding the translations of the texts in a XLIFF file, in your `apps/myapp/i18n/` directory, as explained in the i18n chapter.

How to locate a file

Overview

Some scripts in your applications may need to access files without necessarily knowing where they are. If you were using a bash command, you could use `find` to find them. In symfony, you can do it just as easily with the `sfFinder` class. Doing a complex search is just a matter of adding new search criteria, and the result is a simple array if file paths.

The `sfFinder` class

The `sfFinder` class if a file finder class based on the Perl File::Find::Rule module. It can find either files or directories (or both), and filters the search by a set of user-defined *rules*. The basic usage is the following:

1. Create a `sfFinder` object for your search by calling the class method `::type()`. You must precise what kind of result you expect (either `file`, `dir` or `any`)

   ```
   $finder = sfFinder::type('file');
   ```
2. Add rules to refine your search and decrease the number of results

   ```
   $finder = $finder->name('*.php');
   ```
3. Launch the search by calling the `->in()` method, setting the root directory of the search as argument

   ```
   $files = $finder->in('/home/production/myproject');
   ```

All these method calls can be chained to one single line, which is often easier to read:

```
$files = sfFinder::type('file')->name('*.php')->in('/home/production/myproject');
// can be read as
// find files with name matching '*.php' in the '/home/production/myproject' directory
```

The `->in()` method returns an array of files, that can easily be used for file manipulation:

```
foreach ($files as $file)
{
  $handle = fopen($file, "r");
  ...
}
```

> **Note**: The `sfFinder` class is autoloaded and doesn't need to be required in your scripts.

Rules principle

The rules used to refine the search are written as method calls of an `sfFinder` object. All methods return the current `sfFinder` object to allow easy chaining.

```
$finder1 = sfFinder::type('file')->name('*.php');                  // is a sfFinder object
$finder2 = sfFinder::type('file')->name('*.php')->size('> 10K');   // is also a sfFinder object
$files = $finder1->in('/home/production/myproject');               // is an array of file paths
```

All rules may be invoked several times, except for the `->in()` method.

Some rules are cumulative (`->name()` for example) whereas others are destructive (like `->maxdepth()`). For destructive rules, only the most recent method call counts:

```
// this one will filter for file names satisfying both conditions
$finder = sfFinder::type('file')->name('*.php')->name('*Success.*');
// same as
$finder = sfFinder::type('file')->name('*Success.php');

// here, only the last call is taken into account
$finder = sfFinder::type('file')->maxdepth(5)->maxdepth(3);
// same as
$finder = sfFinder::type('file')->maxdepth(3);
```

Filter rules

Filter by name

To filter the results on file names, add calls to the `->name()` method with patterns in glob or regular expression format:

```
$finder = sfFinder::type('file')->name('*.php');
$finder = sfFinder::type('file')->name('/.*\.php/');
```

You can even exclude certain file names from the result, doing negative filtering with the `->not_name()` method:

```
$finder = sfFinder::type('file')->not_name('Base*');
$finder = sfFinder::type('file')->name('/^Base.*$/');
```

Filter by size

You can filter your search on file size by calling the `->size()` method, which expects a string containing a comparison as argument. The method also understands magnitudes:

```
// search only for files bigger than 10 kilobytes
$finder = sfFinder::type('file')->size('> 10K');
// search only for files smaller than 1 kilobyte, or exactly that
$finder = sfFinder::type('file')->size('<= 1Ki');
// search only for files being 123 bytes of size
$finder = sfFinder::type('file')->size(123);
```

The symbols used for magnitude are the binary prefix defined by the International System of Units.

Limiting the search depth

By default, a search made by the `sfFinder` object is recursive and scans all the subdirectories. You can override this default behaviour by using the `->maxdepth()` method to set the maximum depth of search in the file tree structure:

```
// search in directory and subdirectories
$finder = sfFinder::type('file');
// search only in the directory passed to the ->in() method,
// and not in any subdirectory
$finder = sfFinder::type('file')->maxdepth(1);
```

Of course, you can also specify a minimum depth by calling the `->mindepth()` method.

By default, the minimum depth is 0 and the maximum depth is infinite (or close to).

Excluding directories

If you want to exclude directories from the search, you can use two methods:

- the `->prune()` method stops the search in the part of the tree structure where the pattern given as argument is found. See it as an interdiction to go and see what's in a directory:

  ```
  // ignore the content of '.svn' folders
  $finder = sfFinder::type('any')->prune('.svn');
  ```

  The finder doesn't go deeper in any of the `.svn` folders, but the `.svn` folders themselves are still part of the results.
- the `->discard()` method removes the files or folders that match the argument from the result, but doesn't stop the tree structure exploration.

  ```
  // remove the '.svn' folders from the result
  $finder = sfFinder::type('any')->discard('.svn');
  ```

These two methods are often used in conjunction, when a directory *and* its content need to be excluded from a search:

```
// remove the '.svn' folders and their content from the result
$finder = sfFinder::type('any')->prune('.svn')->discard('.svn');
```

Search starting point

The `->in()` method is used to specify where the `sfFinder` has to look for files or directories. It can take a file path or an array of file paths as argument:

```
// search in a single location
$files = $finder->in('/home/production/myproject');
// search in several locations
$files = $finder->in(array('/home/production/myproject', '/home/production/myotherproject'));
```

It can accept either absolute or relative paths:

```
// absolute path
$files = $finder->in('/home/production/myproject');
// relative path
$files = $finder->in('../projects/myproject');
```

Returning relative paths

By default, the paths returned by the `->in()` method are absolute paths. You can choose to receive an array of relative paths in place, by chaining the call to the `->relative()` method before calling `->in()`:

```
// paths results are relative to the root directory
$files = $finder->in('/home/production/myproject');
// paths results are relative to the current directory,
// i.e. the directory of the current script
$files = $finder->relative()->in('/home/production/myproject');
```

How to use plug-ins

Overview

If you want to extend symfony, the plug-in system ensures you a great flexibility for distribution, upgrade and clean separation with the rest of the framework. Based on the PEAR extension, the symfony plug-in system allows to distribute custom libraries or modules.

What is a plug-in?

A plug-in is a PEAR package. This has several benefits:

- Install, upgrades and uninstall are built-in features
- Dependencies are also managed by PEAR
- It natively supports various ways to broadcast a plug-in (from a local archive, from a URL, or from a PEAR channel)
- The discovery of new plug-ins is automatic

You will find more about the PEAR distribution system in the PEAR website.

The symfony plug-in system takes advantage of all the features of PEAR, and adds on top of that a custom set of roles, or locations, which allow a plug-in to be installed globally (it will then be available for all your PHP scripts) or locally (it is then copied in a specific project tree structure, and is only available there).

Installing a plug-in

The symfony command line makes it easy to install a plug-in. For instance, to install the `sfShoppingCart` plug-in of the `symfony` channel for the current project, type:

```
$ symfony plugin-install local symfony/sfShoppingCart
```

This will install the shopping cart classes in the project `lib/` directory, and they will be available to the whole project without the need to require them manually (classes located in the the `lib/` directory are autoloaded.

> **Note**: As the symfony plug-in system relies on PEAR, the `plugin-install` will only work if you have PEAR 1.4.0 or higher installed.

If you know that you will need the plug-in for various projects, it is better to install it globally:

```
$ symfony plugin-install global symfony/sfShoppingCart
```

In this case, the library will be installed in the `PEAR_LIB` directory. The command is equivalent to:

```
$ pear install symfony/sfShoppingCart
```

> **Note**: The `global` installation of a plug-in can only work if you installed symfony itself via PEAR (not if you installed symfony from a SVN export or a downloaded archive). This is because the plug-ins specify a minimum version for the symfony framework, and this version is given by PEAR.

The plug-in location is not limited to a PEAR channel. You can also use a local .tgz archive or point directly to a web URL:

```
$ symfony plugin-install global /home/production/downloads/sfShoppingCart.tgz
$ symfony plugin-install global http://pear.symfony-project.com/get/sfShoppingCart-1.0.0.tgz
```

The symfony command line also provides other commands:

Command name	Effect
`plugin-uninstall`	uninstall a plug-in
`plugin-upgrade`	upgrade a plug-in
`plugin-upgrade-all`	upgrade all plug-ins

Some plug-ins contain a whole module. Once again, a module plug-in can be installed either locally or globally. The only difference between module plug-ins and classic modules is that module plug-ins don't appear in the `myproject/apps/myapp/modules/` directory (to keep them easily upgradeable) and need to be activated in the `settings.yml` file:

```
all:
  .settings:
    activated_modules:  [default, sfMyPluginModule]
```

This is to avoid that a plug-in module becomes available by mistake for a project that doesn't require it, which could open a security breach in symfony applications.

Anatomy of a plug-in

A plug-in comes under the shape of a .tgz package, containing at least a `package.xml` file and a folder with the files of the plug-in.

For instance, let's have a look at the `sfShoppingCart` plug-in. You can download the archive at:

```
http://pear.symfony-project.com/get/sfShoppingCart-1.0.0.tgz
```

The `package.xml` follows the PEAR syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.6" version="2.0" xmlns="http://pear.php.net/dtd/package-2.0" xmlns:
 <name>sfShoppingCart</name>
 <channel>pear.symfony-project.com</channel>
 <summary>symfony shopping cart.</summary>
 <description>symfony shopping cart.</description>
 <lead>
  <name>Fabien POTENCIER</name>
  <user>fabpot</user>
  <email>fabien.potencier@symfony-project.com</email>
```

```
  <active>yes</active>
 </lead>
 <date>2006-01-18</date>
 <time>15:54:35</time>
 <version>
  <release>1.0.0</release>
  <api>1.0.0</api>
 </version>
 <stability>
  <release>stable</release>
  <api>stable</api>
 </stability>
 <license uri="http://www.symfony-project.com/license">MIT license</license>
 <notes>-</notes>
 <contents>
  <dir name="/">
   <file baseinstalldir="symfony/plugins/sfShoppingCart" install-as="sfShoppingCart.class.php" md
   <file baseinstalldir="symfony/plugins/sfShoppingCart" install-as="sfShoppingCartItem.class.php
  </dir>
 </contents>
 <dependencies>
  <required>
   <php>
    <min>5.0.0</min>
   </php>
   <pearinstaller>
    <min>1.4.1</min>
   </pearinstaller>
   <package>
    <name>symfony</name>
    <channel>pear.symfony-project.com</channel>
    <min>0.6.0</min>
    <max>0.8.0</max>
   </package>
  </required>
 </dependencies>
 <phprelease />
 <changelog />
</package>
```

The interesting parts here are the `<contents>` and the `<dependencies>` tags. For the rest of the tags, there is nothing specific to symfony, so you can refer to the PEAR online manual.

Contents

A plug-in is a set of files that can contain several types of data. According to the type of data - or *role* - defined in the related `<content>` tag, the file will be copied in a different location on the disk.

Role	Location for local	Location for global
php	myproject/lib/	PEAR_PHP_DIR/
data	myproject/data/	PEAR_DATA_DIR/
bin	myproject/batch/	PEAR_BIN_DIR/
test	myproject/test/	PEAR_TEST_DIR/
doc	myproject/doc/	PEAR_DOC_DIR/

> **Note**: To get the value of the PEAR directories in your installation, type in the command line:

```
$ pear config-show
```

The `baseinstalldir` attribute tells to PEAR where to copy the content referenced in the `<content>` tag.

For instance, the 2 files of the `sfShoppingCart` plug-in are installed as follow:

File	Local install
`sfShoppingCart.class.php`	`myproject/lib/symfony/plugins/sfShoppingCart.class.`
`sfShoppingCartItem.class.php`	`myproject/lib/symfony/plugins/sfShoppingCartItem.cl`

The contents defined with a `php` role have the great advantage of being autoloaded whether being installed locally or globally.

In addition, the choice to specify a basic tree structure pattern like `symfony/plugins/pluginName` for plug-ins ensures that they won't be lost in the PEAR of lib/ directories.

Dependencies

Plug-ins are designed to work with a given set of versions of PHP, PEAR, symfony and sometimes other foreign libraries (like Propel) or plug-ins. Declaring these dependencies in the `<dependencies>` tag tells PEAR to check that the required packages are already installed, and to raise an exception if not.

When installing a local plug-in, the dependencies are not installed if not found. When installing a global plug-in, the PEAR configuration is used to see if the dependencies have to be installed when necessary, so beware that installing a plug-in requiring a newer version of symfony could launch the upgrade of the framework.

Building your own plug-in

If you decide to write a plug-in of your own, there are a few guidelines and rules to follow.

Naming conventions

To keep the PEAR and `lib/` directories clean, we recommend that all the plug-in names are in camelCase and start with 'sf' (ex: 'sfShoppingCart', 'sfFeed', etc.). Before naming your plug-in, check that there is no existing PEAR package or symfony plug-in with the same name.

> **Note**: Plug-ins relying on Propel should contain 'Propel' in the name. For instance, an authentication plug-in using the Propel data access objects should be called 'sfPropelAuth'.

For a library, the files of role `php` should have a base install directory starting with `symfony/plugins/pluginName/`. For a module, the files should be of role `data` and have a base install directory starting with `symfony/modules/pluginName`.

Included files

Plug-ins should include a `LICENSE` file describing the conditions of use and the chosen license. You are also advised to add a `README` file to explain the version changes, interest of the plug-in, effect, installation and configuration instructions, etc.

The files to install should be placed in directories according to their role. The `php` files should be in a `lib/` folder, the `data` files in a `data/` folder, etc.

And, of course, you have to write the `package.xml` file. Don't bother to add the `md5sum` to the `<content>` tags though, the PEAR package builder will do it for you (see below). To have an idea about how a typical pre-build `package.xml` looks like, check the ones in the SVN repository.

All in all, your plug-in directory should include the following files:

```
LICENSE
README
package.xml
lib/
  file1.php
  ...
data/
  ...
```

Most of the time, plug-ins should be able to work even if the archive is simply extracted in a `lib/` directory. Keep that in mind when designing the file structure of your own plug-in.

Post install script

A PEAR package can include a post install script, which will manage installation operations that cannot be done with the standard PEAR mechanisms. The symfony plug-ins have access to the same feature. This allows you to setup a configuration when installing a module, add some data to a database, copy files out of the usual directories, etc.

If you want to learn more about the PEAR post-install scripts, refer to the PEAR manual.

Modules and embedded files

If your plug-in is a module, it has to embed all the necessary configuration files with it. Use the classical file structure of a module to include a `module.yml` file for instance, and all i18n materials should definitely go into a `i18n/` directory.

If your module has a specific data model, written in a `schema.yml`, you will need a post install script to copy it to the project `config/` directory for local installation. For global installations, your README file will have to specify that the `schema.yml` is to be copied manually in the `config/` directory of each project using the plug-in. In both cases, you must not override any existing `schema.yml` in the projects of the user, so it is best to call your schema `pluginName_schema.yml` (as soon as the name ends with `schema.yml`, the `propel-build-model` command will take it into account).

If your plug-in includes media files (images, css, javascripts, etc.), you should declare them with a `data` role and specify a base install directory like `symfony/web/sf/MEDIA_TYPE/pluginName/`. For instance, to store a `myimage.jpg` images for the `sfMyPlugin` plug-in, write the `<content>` tag as follows:

```
<file baseinstalldir="symfony/web/sf/img/sfMyPlugin" install-as="myimage.jpg" name="img/myimage.j
```

Once you do a global install of the plug-in, as you already have a `sf/` symbolic link to `symfony/web/sf/` in your server configuration, the image can be addressed to as:

```
<?php echo image_tag('/sf/img/sfMyPlugin/myimage.jpg') ?>
```

The trouble is for local install. The image will be installed in `myproject/data/symfony/web/sf/img/sfMyPlugin/myimage.jpg`, and this path is not accessible to the web server. The other problem is that you cannot copy the files to a local `web/sf/` directory, since the server already has an alias for that.

The solution is to add a post install script to your PEAR package that will copy manually the media files to a `web/img/sfMyPlugin/` directory and remove the reference to the `sf/` directory in the case of a local install.

Dependencies

You should always declare dependencies on PHP, PEAR and symfony, at least the ones corresponding to your own install, as a minimum requirement. If you don't know what to put, add a requirement for PHP 5.0, PEAR 1.4 and symfony 0.6 (plug-ins are not supported before 0.6).

We also recommend that you add a maximum version of symfony for your plug-in. This will cause an error message when trying to use your plug-in with a more advanced version of the framework, and this will oblige you to make sure that your plug-in works correctly with this version before releasing it again. It's better to have an alert and to download an upgrade rather than having a plug-in fail silently.

Building the plug-in

The PEAR component has a command that creates the .tgz archive of the package, provided you call the following command from a directory containing a `package.xml`:

```
$ pear package
```

Couldn't be easier.

Once you built your plug-in, check that it works by installing it yourself.

Hosting your plug-in at symfony-project.com

The symfony team will be pleased to host your plug-in, provided that you respect the aforementioned rules and two additional ones: - the license of the plug-in must be a MIT one - the author of the plug-in must sign a symfony contributors license agreement

The main advantages of being hosted by symfony are that your plug-in will be visible in the symfony channel, you will have a SVN access to a subdirectory of the `plugin/` repository, and we will automatically build a beta package based on the latest SVN version of your plug-in every night.

How to write a batch process

Overview

Creating batch processes in symfony is very simple. Batch processes can be used to automate any function that might need to be repeated on a regular basis. Some examples of useful batch functions might include a tool to load the development database with test data, or a production tool to periodically expire accounts, or send batched emails at a scheduled time.

Batch skeleton

In symfony, a batch process refers to any script that is run outside of the normal web front controller. To create a batch process, you need to define the location, application and environment that the batch process is to use.

This is easily accomplished by creating a PHP file in the `myproject/batch/` directory and starting it with the following:

```
<?php

define('SF_ROOT_DIR',      realpath(dirname(__file__).'/..'));
define('SF_APP',           'myapp');
define('SF_ENVIRONMENT',   'prod');
define('SF_DEBUG',          false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATO

// initialize database manager
$databaseManager = new sfDatabaseManager();
$databaseManager->initialize();

// batch process here
?>
```

This script does nothing, or close to nothing: It defines a path, an application and an environment to get to a configuration, and loads that configuration. But that's already a lot because it means that all of the code written in your batch process will take advantage of the auto-loading of classes, automatic connection to Propel objects, and the symfony root classes.

> **Note:** If you examine symfony's front controllers (like `web/index.php`) you will find this code extremely familiar. That's because every web request requires access to the same objects and configuration as a batch process.

The value that you define for `SF_APP` could be the name of any of the applications that you have defined in your project.

The `SF_ENVIRONMENT` has to be initialized with the name of the environment you want to run the batch in. The default possible environments are `prod`, `dev`, and `test`.

The `SF_DEBUG` constant defines whether the configuration has to be parsed every time the batch is run (`true`), or if a cached version can be used to imrpove speed (`false`). In general, the debug mode is used only in development. You can read more about the debug mode in the debug chapter.

Example batch processes

Below are some examples of batch processes that you might create.

Loading Test Data

Loading test data in the development and/or test environments is something that happens repeatedly. A PHP batch can automate this process. First, define the data to be loaded in a YAML file with the following format:

```
TableName:
  recordlabel:
    column:  value
    column:  value
```

Save one or more data files with this format in a `myproject/data/fixtures/` directory. You can find more about how to create the data population files in the data files chapter.

The following script excerpt reads all YAML files in the `fixtures/` directory and loads the data into the database defined in the `app/config/databases.yml` configuration file.

```
// batch process here
$data = new sfPropelData();
$data->loadData(sfConfig::get('sf_data_dir').DIRECTORY_SEPARATOR.'fixtures');
```

Processing

TODO

- Create additional examples
- Scheduling batches

Command Line Interface

Overview

Many of the tasks that a developer executes during the building and maintenance of a web application can be handled by symfony's Command Line Interface (CLI).

Pake

Symfony uses a dedicated tool called Pake to manage common tasks. Pake is a php tool similar to the Rake command, a Ruby translation of the `make` command. It automates some administration tasks according to a specific configuration file called `pakefile.php`. But since you use the `pake` tool just by typing `symfony` in a command line, everything becomes much simpler than it sounds.

> **Note**: The symfony CLI works only from the root of a symfony project

CLI core

To get the installed version of the symfony package, type:

```
$ symfony -V
```

To get the full list of the available administration operations, use the `-T` option or symply type the `symfony` command with no argument:

```
$ symfony
```

The symfony command expects tasks, and some tasks require additional parameters. The general syntax is:

```
$ symfony <TASK> [parameters]
```

A few tasks have a shortcut, faster to write, that has the same effect.

```
$ symfony cc
// does the same thing as
$ symfony clear-cache
```

When an exception occurs, you might want to get the stack trace and detailed explanation. Add the `-t` option before the task name to get the trace.

CLI tasks

Structure generation

`init-project`: Initialize a new symfony project (shortcut: `new`)

```
$ symfony init-project <PROJECT_NAME>
```

`init-app`: Initialize a new symfony application (shortcut: `app`)

```
$ symfony init-app <APPLICATION_NAME>
```

`init-module`: Initialize a new symfony module (shortcut: `module`)

```
$ symfony init-module <APPLICATION_NAME> <MODULE_NAME>
```

`init-batch`: Initialize a new batch file (shortcut: `batch`). You must select a batch skeleton to init, and then follow the prompts

```
$ symfony init-batch <SKELETON_NAME> [...]
```

`init-controller`: Initialize a new controller (shortcut: `controller`). The default script name follows the symfony convention.

```
$ symfony init-controller <APPLICATION_NAME> <ENVIRONMENT_NAME> [<SCRIPT_NAME>] [true|false]
```

Find more about these commands in the project creation chapter.

Model generation

`propel-build-model`: Create Propel classes for the current model, based on the schema files (YAML or XML) of your `config/` directory.

```
$ symfony propel-build-model
```

The connection settings used by the following commands are taken from the `config/propel.ini` configuration.

`propel-build-schema`: Create `schema.xml` from an existing database

```
$ symfony propel-build-schema
```

`propel-build-sql`: Create the SQL code to create the tables described in the `schema.yml`, in a `data/schema.sql` file.

```
$ symfony propel-build-sql
```

`propel-build-db`: Create an empty database based on the connection settings

```
$ symfony propel-build-db
```

`propel-insert-sql`: Insert the SQL code from `data/schema.sql` into the database

```
$ symfony propel-insert-sql
```

`propel-build-all`: Executes `propel-build-schema`, `propel-build-sql` and then `propel-insert-sql` all in one command.

```
$ symfony propel-build-all
```

`propel-load-data`: Loads all data from default directory `data/fixtures` unless otherwise specified. Environment is default to `dev`. The fixtures directory must be specified relative to the project's data dir, for example `fixtures` (default) or `testdata` or specify a single file `fixtures/file.yml`.

```
$ symfony propel-load-data  <APPLICATION_NAME> [<ENVIRONMENT_NAME>] [<FIXTURES_DIR_OR_FILE>]
```

`propel-build-all-load`: Executes `propel-build-all` then `propel-load-data`. Accepts same arguments as `propel-load-data`.

```
$ symfony propel-build-all-load  <APPLICATION_NAME> [<ENVIRONMENT_NAME>] [<FIXTURES_DIR_OR_FILE>]
```

If you want to learn more about the model and the effect of these commands, refer to the model chapter.

Development tools

`clear-cache`: Clear the cached information (shortcut: `cc`) (find more in the cache chapter)

```
$ symfony clear-cache <APPLICATION_NAME> [template|config]
```

`clear-controllers`: Clear the web directory of all controllers other than ones running in a production environment. Very useful before deployment to the production server.

```
$ symfony clear-controllers
```

`fix-perms`: Fix directories permissions, to change to `777` the directories that need to be writable. The permission can be broken if you use a checkout from a SVN repository.

```
$ symfony fix-perms
```

`test`: Launch the test suite for an application (find more in the unit test chapter)

```
$ symfony test <APPLICATION_NAME>
```

`freeze`: Copies all the necessary symfony libraries into the `data/`, `lib/` and `web/sf/` directories of your project. Your project then becomes a kind of sandbox, i.e. a standalone application with no dependence and ready to be transferred to production via FTP. Works fine with PEAR installations as well as symbolic links. Unfreeze your project with the `unfreeze` task.

```
$ symfony freeze
$ symfony unfreeze
```

`sync`: Synchronise the current project with another machine (find more in the deployment chapter)

```
$ symfony sync <ENVIRONMENT_NAME> [go]
```

Project administration

`disable`: Forwards the user to the unavailable module and action in your `settings.yml` file and acts in the same way as if you had set the unavaiable setting in your `settings.yml` file. The advantage over the setting is that you can disable a single application for a single environment, and not only the whole project.

```
$ symfony disable <APPLICATION_NAME> <ENVIRONMENT_NAME>
```

`enable`: Enables the application and clears the cache

```
$ symfony enable <APPLICATION_NAME> <ENVIRONMENT_NAME>
```

`purge-logs: This clears the logs files in the log directory depending on your settings in logging.yml (find more in the logging chapter)

```
$ symfony purge-logs
```

`rotate-log`: Forces a rotation of a log file if rotate is enabled for the log file in logging.yml. (find more in the logging chapter)

```
$ symfony rotate-log <APPLICATION_NAME> <ENVIRONMENT_NAME>
```

Scaffolding and admin generation

`propel-generate-crud`: Generate a new Propel CRUD module based on a class from the model

```
$ symfony propel-generate-crud <APPLICATION_NAME> <MODULE_NAME> <CLASS_NAME>
```

`propel-init-crud`: Same as `propel-generate-crud`, except that the generated code is visible only in the `cache/` folder (the generated actions and templates inherit from the framework)

`propel-init-admin`: Initialize a new Propel admin module based on a class from the model

```
$ symfony propel-init-admin <APPLICATION_NAME> <MODULE_NAME> <CLASS_NAME>
```

You will find a lot of details about scaffolding and generated administrations in the scaffolding and generator chapter.

Plugin management

`plugin-install`: Install a new plugin

```
$ symfony plugin-install [local|global] <CHANNEL_NAME>/<PLUGIN_NAME>
```

`plugin-upgrade`: Upgrade a plugin

```
$ symfony plugin-upgrade [local|global] <CHANNEL_NAME>/<PLUGIN_NAME>
```

`plugin-upgrade-all`: Upgrade all the plugins previously installed in local

```
$ symfony plugin-upgrade-all
```

`plugin-uninstall`: Uninstall a plugin

```
$ symfony plugin-uninstall [local|global] <CHANNEL_NAME>/<PLUGIN_NAME>
```

The way to build, install and manage plugins is described in the plugin chapter.

Fast Web server for test

The `server` command launches a fast web server written in PHP to serve a symfony application in the `dev` environment. The default port is 8000.

```
$ symfony server <APPLICATION_NAME> [<PORT>]
```

For instance, if you want to test a symfony application called `myapp` without changing your server configuration, type:

```
$ symfony server myapp
```

You can now browse your app in the `dev` environment by requesting

```
http://localhost:8000/
```

Symfony uses nanoserv for this purpose, in order to bypass completely the web server. This results in a very fast access to the development environment - without any control over the server configuration nor access to its log files. This tool is to be used for debug, not in a production environment.

> **Note**: This feature is still at an early stage

Automatic completion

The symfony wiki contains user-contributed configuration files to allow automatic completion of symfony commands. Check out the one that fits your CLI:

- Bash completion
- Zsh completion

How to upload a file

Overview

Backends and collaborative applications often require users to upload media or data files. With a few lines of code, symfony handles all that needs to be taken care of - file renaming, move to upload directory, etc. And the users of the admin generator also have access to a new helper that reduces the implementation to a simple configuration.

Regular file upload

Uploading a file requires a form in a template, and an action to handle it. For the template, use the `input_file_tag()` helper in a form declared as `multipart`:

```
<?php echo form_tag('media/upload', 'multipart=true') ?>
  <?php echo input_file_tag('file') ?>
  <?php echo submit_tag('Send') ?>
</form>
```

It generates the following HTML code:

```
<form method="post" enctype="multipart/form-data" action="media/upload">
  <input type="file" name="file" id="file" value="" />
  <input type="submit" name="commit" value="Send" />
</form>
```

The action (`media/upload` in this example) moves the file from the request to the upload directory:

```
public function executeUpload()
{
  $fileName = $this->getRequest()->getFileName('file');

  $this->getRequest()->moveFile('file', sfConfig::get('sf_upload_dir').'/'.$fileName);

  $this->redirect('media/show?filename='.$fileName);
}
```

The `sf_upload_dir` parameter holds the absolute path, in your server, where the file is to be uploaded. For a project called `myproject`, it is usually `/home/production/myproject/web/upload/`. You can modify it easily in the `config/config.php`:

```
sfConfig::add(array(
  'sf_upload_dir_name'  => $sf_upload_dir_name = 'uploads',
  'sf_upload_dir'       => sfConfig::get('sf_root_dir').DIRECTORY_SEPARATOR.sfConfig::get('sf_web
));
```

> **Note**: Before moving the file from the request to the upload directory, you should sanitize the filename to replace special characters and avoid file system problems. In addition, you should escape the `$fileName` before passing it as a request parameter.

To display the uploaded file, use the `sf_upload_dir_name` parameter. For example, if the uploaded media is an image, the `media/show` template will display it with:

```
...
<?php echo image_tag('/'.sfConfig::get('sf_upload_dir_name').'/'.$sf_params->get('filename')) ?>
```

Validation

Like regular form inputs, file upload tags can be validated by a symfony validator, the `sfFileValidator`. Just remember to put a custom `file: true` parameter in the validator declaration under the `names` key. For instance, the `validate/upload.yml` for the previous form should be written as follows:

```
methods:
  post:               [file]

names:
  file:
    required:         Yes
    required_msg:     Please upload a file
    validators:       myFileValidator
    file:             true

myFileValidator:
  class:              sfFileValidator
  param:
    mime_types:
      - 'image/jpeg'
      - 'image/png'
      - 'image/x-png'
      - 'image/pjpeg'
    mime_types_error: Only PNG and JPEG images are allowed
    max_size:         512000
    max_size_error:   Max size is 512Kb
```

> **Note**: Due to inconsistencies between Internet Explorer and other browsers, you have to use the IE-specific mime types as well as the regular ones.

The `sfFileValidator` can validate the type of the uploaded file (you can specify an array of mime types) its size (you can specify a minimum and a maximum size).

Thumbnails

If you upload images, you might need to create thumbnails of each uploaded file. In this case, the `sfThumbnail` plugin might prove useful.

First, install the plugin using the symfony command line:

```
$ symfony plugin-install local symfony/sfThumbnail
```

> **Note**: If the GD library is not activated, you might have to uncomment the related line in your `php.ini` and restart your web server to enable PHP image handling functions.

Once the plugin is installed, you can use it through the `sfThumbnail` object. For instance, to save a thumbnail of maximum size 150x150px at the same time as the uploaded image in the above example, replace the `media/upload` action by:

```
public function executeUpload()
{
  $fileName = $this->getRequest()->getFileName('file');

  $thumbnail = new sfThumbnail(150, 150);
  $thumbnail->loadFile($this->getRequest()->getFilePath('file'));
  $thumbnail->save(sfConfig::get('sf_upload_dir').'/thumbnail/'.$fileName, 'image/png');

  $this->getRequest()->moveFile('file', sfConfig::get('sf_upload_dir').'/'.$fileName);

  $this->redirect('media/show?filename='.$fileName);
}
```

Don't forget to create the `uploads/thumbnail/` directory before calling the action.

File upload with the admin generator

If your data model contains a field for a media, and if your administration is built using the admin generator, you can use the long named `object_admin_input_upload_tag()` helper. It does everything for you with just a simple configuration.

For instance, if you have a `User` table with a `photo` column, the `generator.yml` for the `edit` view can be configured as follows:

```
edit:
  title:          User profile
  fields:
    photo:
      name:       User photo
      help:       max width 200px
      type:       admin_input_upload_tag
      upload_dir: pictures/users
      params:     include_link=pictures/users include_remove=true
  display:        [name, email, photo]
```

The `upload_dir` key sets the upload directory (under the `uploads/` directory).

If you include an `include_link` param, a link will be added to the uploaded media (that is, if a media is uploaded). The text of the link is '[show file]' by default - unless you specify an `include_text` param.

If you include an `include_remove` param, the helper will display a link to allow easy removal of the media when clicked.

How to manage a shopping cart

Overview

Symfony offers a plugin to manage shopping carts in ebusiness websites. Adding an item, changing quantities and displaying the content of a shopping cart is made easy and painless.

Installation

The shopping cart classes are packaged into a `sfShoppingCart` plugin. They are not shipped with the default symfony installation. Symfony plugins are installed via PEAR (find more about plugins in the related page).

To install the shopping cart classes globally for all your projects, type in the command line:

```
$ symfony plugin-install global symfony/sfShoppingCart
```

If you want to use it only within one project, prefer the `local` install:

```
$ symfony plugin-install local symfony/sfShoppingCart
```

Whether you opt for a local or a global install, the classes will autoload when needed.

Constructor

The `sfShoppingCart` class is aimed to manage shopping carts. It can contain any kind of object.

The constructor allows to declare the tax rate to apply to the shopping cart:

```
$my_shopping_cart = new sfShoppingCart(APP_CART_TAX);
```

In this example, the shopping cart tax rate is written in the `app.yml` configuration file of the application for easy change:

```
all:
  cart:
    tax:  19.6
```

Create a user shopping cart

You can easily create a new `sfShoppingCart` object in an action with the `new` constructor. However, it will not be of any use if it is not linked to a user session. The easiest way to keep a user's choice in a shopping cart is to make a composition of an `sfShoppingCart` object into the `sfUser` class. To do that, add a custom method to the `myproject/apps/myapp/lib/myUser.php` class:

```
class myUser extends sfUser
{
```

```
  public function getShoppingCart()
  {
    if (!$this->hasAttribute('shopping_cart'))
      $this->setAttribute('shopping_cart', new sfShoppingCart(APP_CART_TAX));

    return $this->getAttribute('shopping_cart');
  }

  ...
}
```

The `$user->getShoppingCart()` method will create a new shopping cart if the user doesn't already have one.

> **Note**: if you need more information about the way to override the default `sfUser` class by a custom `myUser` class, you should read the section about **factories** in the configuration chapter.

Add, modify and remove items

The shopping cart can contain any quantity of objects from different classes. Each item stored in the shopping cart is an instance of the `sfShoppingCartItem` class.

The `sfShoppingCart` class has `->addItem()` and `->deleteItem()` methods. As you can add or delete any type of object, the first argument of these method calls is the class name of the object.

To modify the quantity of one item, first get the `sfShoppingCartItem` object itself (via the `->getItems()` method of the `sfShoppingCart` object) and call its `->setQuantity()` method.

The shoppingcart module

Here is a possible module implementation of the shopping cart management where objects of class 'Product' (representing products) can be added, modified or suppressed with the actions 'add', 'update' and 'delete':

```
class shoppingcartActions extends sfActions
{
  ...

  public function executeIndex()
  {
    $this->shopping_cart = $shopping_cart;
    $this->items = $shopping_cart->getItems();

    ...
  }


  public function executeAdd()
  {
    ...

    if ($this->hasRequestParameter('id'))
    {
```

```
      $product = ProductPeer::retrieveByPk($this->getRequestParameter('id'));
      $item = new sfShoppingCartItem('Product', $this->getRequestParameter('id'));
      $item->setQuantity(1);
      $item->setPrice($product->getPrice());
      $shopping_cart = $this->getUser()->getShoppingCart();
      $shopping_cart->addItem($item);
    }

    ...
  }

  public function executeUpdate()
  {
    $shopping_cart = $this->getUser()->getShoppingCart();
    foreach ($shopping_cart->getItems() as $item)
    {
      if ($this->hasRequestParameter('quantity_'.$item->getId()))
      {
        $item->setQuantity($this->getRequestParameter('quantity_'.$item->getId()));
      }
    }

    ...
  }

  public function executeDelete()
  {
    if ($this->hasRequestParameter('id'))
    {
      $shopping_cart = $this->getUser()->getShoppingCart();
      $shopping_cart->deleteItem('Product', $this->getRequestParameter('id'));
    }

    ...
  }


  ...
}
```

Add an item

Let's take a closer look at this code.

To add an item to the shopping cart, you call the `->addItem()` method, passing it a `sfShoppingCartItem` object. This object contains the object class and the unique id of the item to be added, the quantity to be added and the price of the item. This allows the shopping cart to contain objects of any class. For example, you could have a shopping cart containing books and CDs.

The price is stored at this moment to avoid difference of price between the product addition and the checkout if a product price is modified in between in a back-office (or if the cart can be kept between sessions). This also allows to apply price discount according to the amount ordered by the client:

```
if ($quantity > 10)
{
  $item->setPrice($product->getPrice() * 0.8);
```

```
}
else
{
  $item->setPrice($product->getPrice());
}
```

The problem is that you loose the original price if you apply the discount this way. That's why the the `sfShoppingCartItem` object has a `->setDiscount()` method that expects a discount percentage:

```
if ($quantity > 10)
{
  $item->setPrice($product->getPrice());
  $item->setDiscount(20);
}
else
{
  $item->setPrice($product->getPrice());
}
```

Modify an item

To change the quantity of an item, use the method `->setQuantity()` of the `sfShoppingCartItem` object. To delete an item, you can either call the `->deleteItem()` method or change the quantity to 0 by calling `->setQuantity(0)`.

If a user adds the same item (same class and same id) several times, the shopping cart will increase the quantity of the item and not add a new one:

```
$item = new sfShoppingCartItem('Product', $this->getRequestParameter('id'));
$item->setQuantity(1);
$item->setPrice($product->getPrice());
$shopping_cart = $this->getUser()->getShoppingCart();
$shopping_cart->addItem($item);
$shopping_cart->addItem($item);

// same as

$item = new sfShoppingCartItem('Product', $this->getRequestParameter('id'));
$item->setQuantity(2);
$item->setPrice($product->getPrice());
$shopping_cart = $this->getUser()->getShoppingCart();
$shopping_cart->addItem($item);
```

Eventually, you may wonder why the `update` action uses arguments like 'quantity_2313=4' instead of 'id=2313&quantity=4'. As a matter of fact, the way this action is implemented allows the update of multiple article quantities at one time.

Delete an entire shopping cart

To reset the shopping cart, simply call the `->clear()` method of the `sfShoppingCart` instance.

```
$this->getUser()->getShoppingCart()->clear();
```

Display the shopping cart in a template

The action `shoppingcart/index` should display the content of the shopping cart. Let's examine a possible implementation.

Get the content of the shopping cart

Three methods of the `sfShoppingCart` object will help you get the content of a shopping cart:

- `->getItems()`: array of all the `sfShoppingCartItem` objects in the shopping cart
- `->getItem($class_name, $object_id)`: one specific `sfShoppingCartItem` object
- `->getTotal()`: Total amount of the shopping cart (sum of the quantity*price for each item)

Shopping cart items also have a parameter holder. This means that you can add custom information to any item.

For instance, in a website that sells auto parts, the `sfShoppingCartItem` objects need to store the objects added, but also the vehicle for which the part was bought. This can be simply done by adding:

```
$item->setParameter('vehicle', $vehicle_id);
```

> **Note**: you may need a `->getObjects()` method instead of `->getItems()`. This method exists but it relies on the Propel data access layer. As the use of Propel is optional, you might not be able to use it. Learn more about the data access layer in the model chapter.

Pass the values to the template

In order to display the content of the shopping cart, the `index` action has to define a few variables accessible to the template:

```
...
$this->shopping_cart = $shopping_cart;
$this->items = $shopping_cart->getItems();
```

The following example shows a simple `indexSuccess.php` template based on an iteration over all the items of the shopping cart to display information about each of them:

```
<?php if ($shopping_cart->isEmpty()): ?>

  Your shopping cart is empty.

<?php else: ?>

  <?php foreach ($items as $item): ?>
    <?php $object = call_user_func(array($item->getClass().'Peer', 'retrieveByPK'), $item->getId(
    <?php echo $object->getLabel() ?><br />
    <?php echo $item->getQuantity() ?><br />
    <?php echo currency_format($item->getPrice(), 'EUR' ) ?>
    <?php if ($item->getDiscount()): >
       (- <?php echo $item->getDiscount() ?> %)
    <?php endif ?><br />
```

```
  <?php endforeach ?>

  Total : <?php echo currency_format($shopping_cart->getTotal(), 'EUR' ) ?><br />

<?php endif ?>
```

Note that this example uses the Propel data access layer. If your project uses another data access layer, this example might need adaptations.

With or without taxes

By default, all the operations (addition with `$shopping_cart->addItem()`, access with `$item->getPrice()` and `$shopping_cart->getTotal()` use prices **without taxes**.

To get the total amount with taxes, you have to call:

```
$total_with_taxes = $shopping_cart->getTotalWithTaxes()
```

If you need it, the `sfShoppingCart` object can be initialized so that the `add` and `get` methods use the price including taxes:

```
class myUser extends sfUser
{
  public function getShoppingCart()
  {
    if (!$this->hasAttribute('shopping_cart'))
    {
      $this->setAttribute('shopping_cart', new sfShoppingCart(APP_CART_TAX));
    }
    $this->getAttribute('shopping_cart')->setUnitPriceWithTaxes(APP_CART_WITHTAXES);
    return $this->getAttribute('shopping_cart');
  }

  ...
}
```

If `APP_CART_WITHTAXES` is set to `true`, the `$shopping_cart->addItem()` and `$item->getPrice()` methods will use prices with taxes. The `->getTotal()` and `->getTotalWithTaxes()` methods still give the correct results.

Once again, it is a good habit to keep the tax configuration in a configuration file: that's why the example above uses a constant `APP_CART_WITHTAXES` instead of a simple `true`. The `myproject/apps/myapp/config/app.yml` should contain:

```
all:
  cart:
    tax:       19.6
    withtaxes: true
```

If you are unsure of the way taxes are handled, just ask the shopping cart:

```
$uses_tax = $shopping_cart->getUnitPriceWithTaxes();
```

How to make sortable lists

Overview

Many web applications need to offer an interface to order items - think about categories in a weblog, articles in a CMS, wishes in an e-commerce website... The old fashion way of doing it is to offer arrows to move one item up or down in the list. The AJAX way of doing it is to allow direct drag-and-drop ordering with server support. This chapter will describe both ways, together with a few tips on the way to enhance your object model and to do complex queries with Creole.

What you need

Data structure

For this article, the example used will be an undefined `Item` table - name it according to your needs. In order to be sortable, records need at least a `rank` field - no need for a heap here since the sorting will be done by the user, not by the computer. So the data structure (to be written in the `schema.yml`) is simply:

```
propel:
  test_item:
    _attributes: { phpName: Item }
    id:
    name:         varchar(255)
    rank:         { type: integer, required: true }
```

Make sure you build your model once the data structure is defined by typing in a command line interface:

```
$ symfony propel-build-model
```

You will also need a database with the same structure. The fastest way of doing it is to call:

```
$ symfony propel-build-sql
$ symfony propel-insert-sql
```

Extending the model

Before thinking about the user interactions, make sure you have a way to retrieve items by rank, to get the list of items ordered by rank, and to get the current maximum rank, by adding the following methods to the `lib/model/ItemPeer.php`:

```
static function retrieveByRank($rank = 1)
{
  $c = new Criteria;
  $c->add(self::RANK, $rank);
  return self::doSelectOne($c);
}

static function getAllByRank()
```

```
{
  $c = new Criteria;
  $c->addAscendingOrderByColumn(self::RANK);
  return self::doSelect($c);
}

static function getMaxRank()
{
  $con = Propel::getConnection(self::DATABASE_NAME);
  $sql = 'SELECT MAX('.self::RANK.') AS max FROM '.self::TABLE_NAME;
  $stmt = $con->prepareStatement($sql);
  $rs = $stmt->executeQuery();

  $rs->next();
  return $rs->getInt('max');
}
```

These methods will be of great use for both sorting interfaces. If you need more information about the way the Object Model handles database queries in symfony, check out the basic CRUD chapter of the Propel user guide.

There are two more method that needs to be added to the `lib/model/Item.php` class. They won't be needed here, but you will probably need them in a real world application, where you will also add and delete items to your table:

```
public function save($con = null)
{
  // New records need to be initialized with rank = maxRank +1
  if(!$this->getId())
  {
    $con = Propel::getConnection(ItemPeer::DATABASE_NAME);
    try
    {
      $con->begin();

      $this->setRank(ItemPeer::getMaxRank()+1);
      parent::save();

      $con->commit();
    }
    catch (Exception $e)
    {
      $con->rollback();
      throw $e;
    }
  }
  else
  {
    parent::save();
  }
}

public function delete($con = null)
{
  $con = Propel::getConnection(PagePeer::DATABASE_NAME);
  try
  {
    $con->begin();
```

```
    // decrease all the ranks of the page records of the same category with higher rank
    $sql = 'UPDATE '.ItemPeer::TABLE_NAME.' SET '.ItemPeer::RANK.' = '.ItemPeer::RANK.' - 1 WHERE
    $con->executeQuery($sql);
    // delete the item
    parent::delete();

    $con->commit();
  }
  catch (Exception $e)
  {
    $con->rollback();
    throw $e;
  }
}
```

Additions and deletions of records have to be managed carefully for the integrity of the `rank` field, that's why the `save()` and `delete()` methods are to be specialized. Because these methods do complex read/write operations and create a risk of concurrency issues, these operations are enclosed in a transaction (refer to the Propel documentation for more details about transactions in symfony).

Preparing the module

The interactions described in this tutorial will take place in a `item` module. Initialize it by calling (assuming you have a `frontend` application):

```
$ symfony init-module frontend item
```

Make sure your web server configuration is OK by testing the access to this new module via your favorite browser. Here is the URL that you should check if you follow this tutorial with a sandbox:

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

Finally, if you want to test the ordering of items, you will need... items. Create a bunch of test items, either via a CRUD interface or a population file.

Now that everything is ready, let's get started.

Classic sortable list

A classic sortable list is a list for which each item has a control to change its order. First, create the action and template that display the list:

```
// add to modules/item/actions/actions.class.php
public function executeList()
{
  $this->items = ItemPeer::getAllByRank();
  $this->max_rank = ItemPeer::getMaxRank();
}

// create a template listSuccess.php in modules/item/templates/
<h1>Ordered list of items</h1>
<ul>
```

```
<?php foreach($items as $item): ?>
  <li>
    <?php
      echo $item->getName().' ';
      if($item->getRank() > 0):
        echo link_to('Move up ', 'item/up?id='.$item->getId());
      endif;
      if($item->getRank() != $max_rank):
        echo link_to('Move down', 'item/down?id='.$item->getId());
      endif;
    ?>
  </li>
<?php endforeach ?>
</ul>
```

The links to move an item up or down are displayed only when the reordering is possible. This means that the first item cannot be moved further up, and the last item cannot be moved further down. Check that the page displays correctly:

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

Now, it's time to look into the `item/up` and `item/down` action. The `up` action is supposed to decrease the rank of the page given as a parameter, and to increase the rank of the previous page. The `down` action is supposed to increase the rank of the page given as parameter, and to decrease the rank of the following page. As they both do two write operations in the database, these actions should use a transaction.

The two actions have a very similar logic, and if you want to keep D.R.Y., you'd better find a smart way to write them without repeating any code. This is done by adding a `swapWith()` method to the `Item.php` model class:

```
public function swapWith($item)
{
  $con = Propel::getConnection(ItemPeer::DATABASE_NAME);
  try
  {
    $con->begin();

    $rank = $this->getRank();
    $this->setRank($item->getRank());
    $this->save();
    $item->setRank($rank);
    $item->save();

    $con->commit();
  }
  catch (Exception $e)
  {
    $con->rollback();
    throw $e;
  }
}
```

Then, the `up` and `down` actions become pretty simple:

```
public function executeUp()
{
```

```
  $item = ItemPeer::retrieveByPk($this->getRequestParameter('id'));
  $this->forward404Unless($item);
  $previous_item = ItemPeer::retrieveByRank($item->getRank() - 1);
  $this->forward404Unless($previous_item);
  $item->swapWith($previous_item);

  $this->redirect('item/list');
}

public function executeDown()
{
  $item = ItemPeer::retrieveByPk($this->getRequestParameter('id'));
  $this->forward404Unless($item);
  $next_item = ItemPeer::retrieveByRank($item->getRank() + 1);
  $this->forward404Unless($next_item);
  $item->swapWith($next_item);

  $this->redirect('item/list');
}
```

If not for the security checks made by the calls to `forward404Unless()`, these actions could be even simpler, but you have to protect your application against wrong requests - the ones that could be done by typing an URL directly.

The list is now fully functional. Try it by moving items up and down in the list.

AJAX sortable list

Base

Developing a basic AJAX sortable list is not harder than developing a classic one. Most of the job is handled by a special JavaScript helper called `sortable_element()`:

```
// add to modules/item/actions/actions.class.php
public function executeAjaxList()
{
  $this->items = ItemPeer::getAllByRank();
}

// create a template ajaxListSuccess.php in modules/item/templates/
<?php use_helper('Javascript') ?>
<style>
  .sortable { cursor: move; }
</style>
<h1>Ordered list of items - AJAX enabled</h1>
<ul id="order">
  <?php foreach($items as $item): ?>
  <li id="item_<?php echo $item->getId() ?>" class="sortable">
    <?php echo $item->getName() ?>
  </li>
  <?php endforeach ?>
</ul>
<div id="feedback"></div>
<?php echo sortable_element('order', array(
  'url'     => 'item/sort',
```

```
  'update' => 'feedback',
)) ?>
```

Check out the result by typing:

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

By the magic of the `sortable_element()` JavaScript helper, the `<ul>` element is made sortable, which means that its children can be reordered by drag and drop. Every time that the user drags an item and releases it to reorder the list, an AJAX request is made with the following parameters:

```
POST /sf_sandbox/web/frontend_dev.php/item/sort HTTP/1.1
  order[]=1&order[]=3&order[]=2&order[]=4&order[]=5&order[]=6&_=
```

The full ordered list is passed as an array (with the format `order[$rank]=$id`, the `$order` starting at 0 and the `$id` being based on what comes after the underscore (_) in the list element `id` property). The `id` property of the sortable element (`order` in the example) is used to name the array of parameters. The JavaScript helper makes a XMLHttpRequest to the `url` action (`item/sort` in the example), passing the ordered list in POST mode, and uses the result of the action to update the element of id `update` (the `feedback` div in the example)

Handling the AJAX request

Now let us write the `item/sort` action and see how it reorders the list of items:

```
// add to modules/item/actions/actions.class.php
public function executeSort()
{
  $order = $this->getRequestParameter('order');
  $flag = ItemPeer::doSort($order);
  return $flag ? sfView::SUCCESS : sfView::ERROR;
}
```

The ability to reorder the whole list is part of the model, that's why it is implemented a static method of the `ItemPeer` class. Once again, the fact that this method updates many records of the `item` table makes it necessary to enclose the updates in a database transaction.

```
static function doSort($order)
{
  $con = Propel::getConnection(self::DATABASE_NAME);
  try
  {
    $con->begin();

    foreach ($order as $rank => $id)
    {
      $item = ItemPeer::retrieveByPk($id);
      if($item->getRank() != $rank)
      {
        $item->setRank($rank);
        $item->save();
      }
    }
```

```
    $con->commit();
    return true;
  }
  catch (Exception $e)
  {
    $con->rollback();
    return false;
  }
}
```

The value returned by this method will determine which template the action will display. Add the following templates in your `modules/item/templates/` folder:

```
// sortSuccess.php
Ok

// sortError.php
<strong>A problem occurred. Please refresh and try again.</strong>
```

Test the server handling by pressing F5 after rearranging the list. The ordering shouldn't change, proving that the server understood and saved correctly what the AJAX request sent to it.

Focus on the `sortable_element()` options

The Javascript helpers chapter describes the generic options of remote function calls, but this example is a good opportunity to see the ones of the `sortable_element()` in detail.

You can define a **different appearance for hovered list elements** when dragging another element over them with the `hoverclass` parameter:

```
<?php use_helper('Javascript') ?>
<style>
  .sortable { cursor: move; }
  .hovered  { font-weight: bold; }
</style>
...
<?php echo sortable_element('order', array(
  'url'        => 'item/sort',
  'hoverclass' => 'hovered',
)) ?>
```

You can **add non-sortable elements** to the list and restrict the drag-and-drop behaviour to a single class only with the `only` parameter:

```
...
<ul id="order">
  <?php foreach($items as $item): ?>
  <li id="item_<?php echo $item->getId() ?>" class="sortable">
    <?php echo $item->getName() ?>
  </li>
  <?php endforeach ?>
  <li>This element is not part of the ordered list</li>
</ul>
<?php echo sortable_element('order', array(
  'url'    => 'item/sort',
```

```
  'only'   => 'sortable',
)) ?>
```

If the list elements are not displayed vertically like in the previous example, you have to set the `overlap` parameter to `horizontal`:

```
<?php use_helper('Javascript') ?>
<style>
  .sortable { cursor: move; float: left; }
</style>
...
<?php echo sortable_element('order', array(
  'url'     => 'item/sort',
  'overlap' => 'horizontal',
)) ?>
```

If the list to order is not a set of `<li>` elements, you will have to define which child elements of the sortable element are to be made draggable:

```
...
<div id="order">
  <?php foreach($items as $item): ?>
  <div id="item_<?php echo $item->getId() ?>" class="sortable">
    <?php echo $item->getName() ?>
  </div>
  <?php endforeach ?>
  <p>This cannot be dragged</p>
</div>
<?php echo sortable_element('order', array(
  'url'    => 'item/sort',
  'tag'    => 'div',
)) ?>
```

For all AJAX actions, it is good to have a **visual feedback** of background activity and of the success of the request:

```
<div id="feedback"></div>
<div id="indicator" style="display:none;"><img src="/images/activity_indicator.gif" style="displa
<?php echo sortable_element('order', array(
  'url'      => 'item/sort',
  'update'   => 'feedback',
  'loading'  => "Element.show('indicator')",
  'complete' => "Element.hide('indicator')",
  'success'  => visual_effect('highlight', 'feedback'),
)) ?>
```

For more details about these parameters and about some others that are not described here, refer to the script.aculo.us Sortable manual.

Comparison

The two methods are both effective to sort a list, but there are limitations and drawbacks.

For large arrays of items, you will probably need a paginated list. The classic method works fine with a page-by-page list, but the AJAX one needs adaptations, and makes it impossible to rearrange elements outside of their own page. That's why you should probably provide a 'move item to position' feature in addition to the AJAX ordering interface.

The AJAX action is not as well protected against wrong requests as the classic one. In order to avoid any risk of database incoherence, you should add a `validateSort()` method to the `itemActions` class. This method would check that all the items `id`, and only them, are present once and only once in the received array.

One drawback of the `ItemPeer::doSort()` method used in the AJAX sorting is the number of queries it needs to reorder the list. Each movement in a list of n items makes at least n+2 queries to the database. AJAX list are not adapted to long lists, so this may not be a major problem, but if performance is a concern for you, you should refactor this method to have it update the ranks with only one query - for instance using the `UPDATE table SET CASE/WHEN` SQL syntax.

The AJAX interface is definitely more user friendly, especially for long ordering tasks, since there is no obligation of a server refresh between two operations. But the ability to drag elements is new in web interfaces, and users not used to it might find it surprising. Moreover, if you choose the AJAX interface, you will have to think about the size of the draggable elements (they need to be large enough to be grabbed), their aspect, their freedom of movement... A lot of Human-Computer Interaction issues that wouldn't need solving with the classic method.

AJAX interactions are always a problem if your target population may turn JavaScripts off in their browser. This means that in addition to the design of a JavaScript interface, you should then provide the classic interface as an alternative so that your functionality degraces gracefully.

All in all, the AJAX version really feels and looks better, but it is at least twice as long to develop.

How to add a custom extension

Overview

Symfony offers various helpers to accelerate template development (see more in the View chapter). But the need for custom helpers can arise for every project, application, or module. Symfony uses naming conventions to allow the easy inclusion of custom helpers so that they can be used the same way as the standard ones, via a simple function call in the templates. Custom classes are also easy to add thanks to the class auto-loading feature.

Custom helper

Naming conventions

To make custom helpers available to your templates, simply create a PHP file ending with `Helper.php` and place it in a subdirectory of the PHP `include_path` named `helper`. The directory where the helper is placed will determine its availability. For instance:

Helper placed in	Is available for
`lib/helper/`	all applications of the project
`myapp/lib/helper/`	all the modules of the 'myapp' application
`mymodule/lib/helper/`	all the templates of the 'mymodule' module

To use a custom helper in a template, you must first declare it on top of the template, the same way you do for other helpers:

```
<?php use_helper('Name') ?>
```

Example

Let's say that your application uses the wiki syntax to format the data entered by the user. The wiki to HTML conversion may occur quite often, so you will refactor the code in a 'wiki_to_html()' helper function:

Here is the `WikiHelper.php` file stored in the `myapp/lib/helper/` directory:

```
<?php

function wiki_to_html($text)
{
  require_once 'Wiki.class.php';
  $wiki = new Wiki();
  return $wiki->transform($text);
}

?>
```

Here is an example template using the helper:

```
<?php use_helper('Wiki') ?>
```

```
...
<p><?php echo wiki_to_html('Text with a WikiLink.') ?></p>
```

Custom classes

Naming conventions

If you need to extend your actions with a custom class, you can take advantage of the class auto-loading feature. If the class file has the same name as the class itself, symfony will auto-load it when needed, provided it is located in one of the three directories below:

Class placed in	Is available for
`lib/`	all applications of the project
`myapp/lib/`	all the modules of the 'myapp' application
`myapp/modules/mymodule/lib/`	all the templates of the 'mymodule' module

Example

For instance, create a new `myTools.class.php` under the `myproject/lib/` directory:

```
class myTools
{
  public static function stripText($text)
  {
    $text = strtolower($text);

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

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

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

    return $text;
  }
}
```

Now, from any action of your project, you can use the `stripText` method of this class without declaring it previously. Just ask:

```
$my_stripped_text = myTools::stripText($my_text);
```

...and symfony will load the `myTools` class automatically.

Accessing context objects

In a custom helper, shortcuts like `$sf_request` or `$sf_user` won't work. In a custom class, the action-style calls (`$this->getRequest()`, `$this->getUser()`, etc.) won't work either. This is

because these functions and classes are not in a request context. So, in order to access the context objects, you have to use the `sfContext` singleton (`sfContext::getInstance()`). For instance, to get the request parameters, write:

```
$id = sfContext::getInstance()->getRequest()->getParameter('id');
```

Class autoloading

The autoloading feature allows you to create an new object without requiring the file that contains its class definition by hand:

```
$obj = new MyClass();
// If MyClass is autoloaded, this will work without a require
```

The autoloading is configured to work with the `lib/` directories described above, according to the default `autoload.yml` configuration file (found in `$pear_data_dir/symfony/config/autoload.yml`):

```
autoload:

  symfony_core:
    name:           symfony core classes
    ext:            .class.php
    path:           %SF_SYMFONY_LIB_DIR%
    recursive:      on
    exclude:        [vendor]

  symfony_orm:
    name:           symfony orm classes
    files:
      Propel:       %SF_SYMFONY_LIB_DIR%/addon/propel/sfPropelAutoload.php
      Criteria:     %SF_SYMFONY_LIB_DIR%/vendor/propel/util/Criteria.php
      SQLException: %SF_SYMFONY_LIB_DIR%/vendor/creole/SQLException.php
      DatabaseMap:  %SF_SYMFONY_LIB_DIR%/vendor/propel/map/DatabaseMap.php

  symfony_plugins:
    name:           symfony plugins
    ext:            .class.php
    path:           %SF_PLUGIN_DIR%
    recursive:      on

  project:
    name:           project classes
    ext:            .class.php
    path:           %SF_LIB_DIR%
    recursive:      on
    exclude:        [model, plugins, symfony]

  model:
    name:           project model classes
    ext:            .php
    path:           %SF_MODEL_LIB_DIR%

  application:
    name:           application classes
    ext:            .class.php
```

```
    path:           %SF_APP_LIB_DIR%
```

If you want to add other places for symfony to look into, create your own `autoload.yml` file in an application `config/` directory and add in new rules. Each autoloading rule has a label, both for visual organization and ability to override it.

For instance, if you added your own `MyClass` class in `/usr/local/mycustomclasses/`, you could add:

```
mycustomclasses:
   name:           classes that I developed myself
   ext:            .php
   path:           /usr/local/mycustomclasses/
   recursive:      on
```

Third-party libraries

If you need a functionality provided by a third-party class, and if you don't want to copy this class in one of the symfony `lib/` dirs, you will probably install it outside of the usual places where symfony looks for files. In that case, using this class will imply a manual `require` in your code, unless you use the symfony **bridge** to take advantage of the autoloading.

Symfony doesn't (yet) provide tools for everything. If you need a PDF generator, an API to Google Maps or a PHP implementation of the Lucene search engine, you will probably need a few libraries from the Zend Framework. If you want to manipulate images directly in PHP, connect to a POP3 account to read emails, or design a console interface for Linux, you will choose the libraries from eZcomponents.

Fortunately, if you define the right settings, the components from both these libraries will work out of the box in symfony.

The first think that you need to declare (unless you installed the third party libraries via PEAR) is the path to the root directory of the libraries. This is to be done in the application `settings.yml`:

```
.settings:
  zend_lib_dir:   /usr/local/zend/library/
  ez_lib_dir:     /usr/local/ezcomponents/
```

Then, extend the autoload routine by telling which library to look at when the autoloading fails with symfony:

```
.settings:
  autoloading_functions: [[sfZendFrameworkBridge, autoload], [sfEzComponentsBridge, autoload]]
```

What will happen when you create a new object of an unloaded class is:

1. The symfony autoloading function (`Symfony::autoload()`) first looks for a class in the paths declared in the `autoload.yml`
2. If none is found, then the callback functions declared in the `sf_autoloading_functions` setting will be called one after the other, until one of them returns `true`:
    1. `sfZendFrameworkBridge::autoload()`
    2. `sfEzComponentsBridge::autoload()`

3. If these also return `false`, then symfony will throw an exception saying that the class doesn't exist.

Note that this setting is distinct from the rules defined in the `autoload.yml`. The `autoloading_functions` setting specifies bridge classes, and the `autoload.yml` specifies paths and rules for searching.

The available bridges are stored in the `$pear_php_dir/symfony/addon/bridge/` directory.

Plug-ins

Symfony can also be extended through plug-ins. To see how to install, use or create your own plu-in, refer to the related chapter.

How to validate a form

Overview

Form validation can occur on the server side and/or on the client side. The server side validation is compulsory - to avoid wrong data to corrupt a script or a database - and the client side validation is optional, though it greatly enhances the user experience. Symfony automates the server side validation to speed up the development of common web applications.

Base example

Let's illustrate the validation features of Symfony starting with a normal Contact form, without any kind validation, showing the following fields:

- name
- email
- age
- message

In a newly created `contact` module, the first action to write is the one that displays the form by calling the default template:

```
class contactActions extends sfActions
{
  public function executeIndex()
  {
    return sfView::SUCCESS;
  }
}
```

And the corresponding `indexSuccess.php` template contains:

```
<?php echo form_tag('contact/send') ?>
  <label for="name">Name</label> : <?php echo input_tag('name') ?><br />
  <label for="email">Email</label> : <?php echo input_tag('email') ?><br />
  <label for="age">Age</label> : <?php echo input_tag('age') ?><br />
  <label for="message">Message</label> : <?php echo textarea_tag('message') ?><br />
  <?php echo submit_tag() ?>
</form>
```

If you wonder what the `_tag()` functions do, you should probably take a look at the form helpers chapter. The form can now be displayed in the browser by typing the URL `index.php/contact`.

To handle the form submission, the `send` action must be created. For this example, we just need the application to display an "OK" message after submission:

```
class contactActions extends sfActions
{

  ...
```

```
  public function executeSend()
  {
    $this->email = $this->getRequestParameter('email');
  }
}
```

The `sendSuccess.php` template just contains:

```
Your message was sent to our services. The answer will be sent at <?php echo $email ?>
```

You can test the whole process of submitting the form and getting the confirmation, it already works fine. Except that if you try to enter invalid data in the fields, the action may very well crash. The fields do require validation.

Rules

Let's write the validation rules in plain text:

- name: required text field, size must be between 2 and 100 characters
- email: required text field, must contain a valid email address
- age: required number field, must contain a integer between 0 and 120
- message : required field

Symfony can apply these rules almost automatically, provided that you add a new configuration file to the module and change a few details in the template.

Configuration file

By convention, if you want to validate the form data on the call to the `send` action, a configuration file called `send.yml` must be created in the `validate` directory of the module. To validate only the `name` field, you need the following configuration:

```
methods:
  get:            [name]
  post:           [name]

names:
  name:
    required:     Yes
    required_msg: The name field cannot be left blank
    validators:   nameValidator

nameValidator:
  class:          sfStringValidator
  param:
    min:          2
    min_error:    You didn't enter a valid name (at least 2 characters). Please try again.
    max:          100
    max_error:    You didn't enter a valid name (less than 100 characters). Please try again.
```

Let's have a closer look at this file:

First, under the `methods` header, the list of fields to be validated is defined for the method of the form (post by default). To be able to change your mind in the future, you should double the statement for the other method.

Then, under the `names` header, the list of the the fields to be checked, along with their 'required' flag, corresponding error message and mention of any specific validation rules header, are specified.

Eventually, as the 'name' field is declared to have a specific set of validation rules, they are detailed under the corresponding header.

Action modification

The default behavior makes symfony call a predefined `handleError()` method whenever an error is detected in the validation process. This method will simply display the `sendError()` template.

But if you prefer to display the form again with an error message in it, you need to override the default `handleError()` method for the form handling action and end it with a redirection to the `index` action of the `contact` module. Do this by adding the following code to your `contact` actions:

```
class ContactActions extends sfActions
{

  ...

  public function handleErrorSend()
  {
    $this->forward('contact', 'index');
  }
}
```

If you try to fill in the form with this new configuration, and type a wrong name, the form is displayed again, but the data you entered is lost and no error message explains the reason of the failure.

Template modification

To address these two issues, you just need to modify the `indexSuccess.php` template.

Since the *forward* method kept the original request, the template has access to the data entered by the user:

```
<?php echo form_tag('contact/send') ?>
  <label for="name">Name</label> : <?php echo input_tag('name', $sf_params->get('name')) ?><br />
  <label for="email">Email</label> : <?php echo input_tag('email', $sf_params->get('email')) ?><b
  <label for="age">Age</label> : <?php echo input_tag('age', $sf_params->get('age')) ?><br />
  <label for="message">Message</label> : <?php echo textarea_tag('message', $sf_params->get('mess
  <?php echo submit_tag() ?>
</form>
```

> **Note**: You can avoid setting manually the value of the fields from the request by using a special filter. See the **Repopulation** section below.

You can detect whether the form has errors by calling the `->hasErrors()` method of the `sfRequest` object. To get the list of the error messages, you need the method `->getErrors()`. So you should add the following lines at the top of the template:

```
<?php if ($sf_request->hasErrors()): ?>
  <p>The data you entered seems to be incorrect.
  Please correct the following errors and resubmit:</p>
  <ul>
  <?php foreach($sf_request->getErrors() as $error): ?>
    <li><?php echo $error ?></li>
  <?php endforeach ?>
  </ul>
<?php endif ?>
```

Now you may suggest that the field with incorrect data should be highlighted, for instance with a repetition of the error message clearly attached to the label with a â . To that extent, simply add the following line before every field:

```
<?php if ($sf_request->hasError('name')): ?>â   <?php echo $sf_request->getError('name') ?> â  <?
```

Complete configuration file

You can add the other rules to the `send.yml` configuration file to force the validation of all fields. This is what a complete YAML validation file looks like:

```
methods:
  get:      [name, email, age, message]
  post:     [name, email, age, message]

names:
  name:
    required:       Yes
    required_msg:   The name field cannot be left blank
    validators:     nameValidator
  email:
    required:       Yes
    required_msg:   The name field cannot be left blank
    validators:     emailValidator
  age:
    required:       No
    validators:     ageValidator
  message:
    required:       Yes
    required_msg:   The message field cannot be left blank

nameValidator:
  class:          sfStringValidator
  param:
    min:          2
    min_error:    You didn't enter a valid name (at least 2 characters). Please try again.
    max:          100
    max_error:    You didn't enter a valid name (less than 100 characters). Please try again.

ageValidator:
  class:          sfNumberValidator
  param:
```

```
    nan_error:    Please enter an integer
    min:          0
    min_error:    You're not even born. How do you want to send a message ?
    max:          120
    max_error:    Hey, grandma, aren't you too old to surf on the Internet ?

emailValidator:
  class:          sfRegexValidator
  param:
    match:        Yes
    match_error:  "You didn't enter a valid email address (for example: name@domain.com). Please
    pattern:      /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/i
```

Available validators

The available validators can be found in the symfony lib `validator` directory. For the moment, they are:

- `sfStringValidator`: allows you to apply string-related constraints to a parameter

  ```
  nameValidator:
    class:          sfStringValidator
    param:
      min:          2
      min_error:    Please enter a name of at least 2 characters
      max:          100
      max_error:    Please enter a name of at most 100 characters
  ```

- `sfNumberValidator`: verifies if a parameter is a number and allows you to apply size constraints

  ```
  ageValidator:
    class:          sfNumberValidator
    param:
      nan_error:    Please enter an integer
      min:          0
      min_error:    You're not even born. How do you want to send a message ?
      max:          120
      max_error:    "Hey, grandma, aren't you too old to surf on the Internet ?"
  ```

- `sfRegexValidator`: allows you to match a value against a regular expression pattern

  ```
  spamValidator:
    class:          sfRegexValidator
    param:
      match:        Yes
      match_error:  Posts containing more than one http address are considered as spam
      pattern:      /http.*http/i
  ```

  The `match` param determines if the request parameter must match the pattern to be valid (value `Yes`) or match the pattern to be invalid (value `No`)

- `sfEmailValidator`: verifies if a parameter contains a value that qualifies as an email address

  ```
  emailValidator:
    class:          sfEmailValidator
    param:
      email_error:  This email address is invalid
  ```

- `sfCompareValidator`: checks the equality of two different request parameters; very useful for password check

```
methods:
  get:            [password1, password2]
  post:           [password1, password2]


names:
  password1:
    required:     Yes
    required_msg: Please enter a password
  password2:
    required:     Yes
    required_msg: Please retype the password
    validators:   passwordValidator


passwordValidator:
  class:          sfCompareValidator
  param:
    check:        password1
    compare_error: The passwords you entered do not match. Please try again.
```

The `check` param contains the name of the field that the current field must match to be valid.

- `sfPropelUniqueValidator`: validates that the value of a request parameter doesn't already exist in your database. Very useful for primary keys.

```
loginValidator:
  class:          sfPropelUniqueValidator
  param:
    class:        User
    column:       login
    unique_error: This login already exists. Please choose another one.
```

In the example above, the validator will look in the database for a record of class `User` where the column `login` has the same value as the parameter to validate. Note that this validator relies on Propel.

- `sfFileValidator`: applies format (an array of mime types) and size constraints to file upload fields

```
methods:
  post:               [image]


names:
  image:
    required:         Yes
    required_msg:     Please upload an image file
    validators:       imageValidator
    file:             true


imageValidator:
  class:              sfFileValidator
  param:
    mime_types:
      - 'image/jpeg'
      - 'image/png'
      - 'image/x-png'
```

```
      - 'image/pjpeg'
    mime_types_error: Only PNG and JPEG images are allowed
    max_size:         512000
    max_size_error:   Max size is 512Kb
```

Beware that the attribute `file` has to be set to `true` for the field in the `names` section - and that the template must declare the form as `multipart`. Find more information in the file upload chapter.

- `sfCallbackValidator`: Passes the hand to a third party method or function to do the validation (and return true or false).

```
ageValidator:
  class:          sfCallbackValidator
  param:
    callback:      is_integer
    invalid_error: Please enter a number.


creditCardValidator:
  class:          sfCallbackValidator
  param:
    callback:      [myTools, validateCreditCard]
    invalid_error: Please enter a valid credit card number.
```

The callback method or function receives the value to be validated as a first parameter. This is very useful when you don't want to create a full validator class but reuse existing methods of functions.

Repopulation

Note: This feature is only available with a symfony release higher than 1096 and is still considered Alpha, since the interface can change. Feel free to give your feedback about it.

One common concern about forms is the value that the form fields will have when the form is displayed again after a failed validation. If you defined default values, or if you use object helpers, it can be quite tricky to determine how to handle the values from the request. In addition, some controls (namely the checkbox and the select tags) have special ways to pass their value in the request parameters.

Fortunately, symfony takes care of the form repopulation for you. If you want your form to be filled in with the values previously entered by the user, simply add these lines to your validation file:

```
fillin:
  activate: on   # activate the form repopulation
  param:
    name: test   # name of the form
```

This forces you to give a `name` attribute to your form, but it opens the possibility to repopulate a form in a page that contains more than one.

The repopulation works for text and hidden inputs, textareas, radiobuttons, checkboxes and selects (simple and multiple).

You might want to transform the values entered by the user before putting them in a form input. Escaping, url rewriting, transformation of special characters into entities, etc., all the transformations that can be called

through a function (existing or defined by you) can be applied to the fields of your form if you define the transformation under the `converters:` key:

```
fillin:
  activate: on
  param:
    name: test
    converters:          # converters to apply
      htmlentities:      [first_name, comments]
      htmlspecialchars: [comments]
```

The repopulation feature is based on a filter called `sfFillInFormFilter`, and it means that you can take advantage of form repopulation even if you don't use the symfony validation files. To activate the filter, just add it to your `filters.yml` as you would do with a normal filter (see more in the filter chapter).

Complex validation needs

Custom validator

Each `Validator` is a particular class that can have certain parameters. If the validation classes shipped with Symfony are not enough for your needs, you can easily create new ones. Here is the example of a validation class for email addresses, the actually existing `sfEmailValidator`:

```
class sfEmailValidator extends sfValidator
{
  public function execute (&$value, &$error)
  {
    if (!preg_match('~^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$~i', $
    {
      $error = $this->getParameter('email_error');
      return false;
    }
    return true;
  }

  public function initialize ($context, $parameters = null)
  {
    // initialize parent
    parent::initialize($context, $parameters);

    // set defaults
    if (!$this->hasParameter('email_error')
    {
      $this->setParameter('email_error', 'Invalid input');
    }

    return true;
  }
}
```

Now the last part of the `send.yml` file can be replaced with:

```
emailValidator:
  class:          sfEmailValidator
```

```
  param:
   email_error:   You didn't enter a valid email address (for example: name@domain.com). Please t
```

If you need to create a new validation class and if it is a generic one, you should ask to have it included in the framework.

Validate method

Sometimes the power of validators is not enough. This happens mostly when you need to perform a complex validation that relies on context dependent variables. In that case, you can add a new method to your `Action` class named `validateXXX()` where `XXX` is the name of the action called by the form. If this method returns `true`, the `executeXXX()` method will be evaluated as usual. Otherwise, it's the `handleErrorXXX()` (if it exists), or the general `handleError()` method that is evaluated.

```
class ContactActions extends sfActions
{

  ...
  public function executeIndex()
  {
    // display the form
    ...
  }

  public function validateSend()
  {
    // validate request parameters
    $spammer = SpammersPeer::retrieveByName($this->getRequestParameter('name'));
    if($spammer && $spammer->isBanned())
    {
      $this->getRequest()->setError('time', 'You are not allowed to post here anymore');
      return false;
    }
    return true;
  }

  public function executeSend()
  {
    // handle the form submission
    ...
  }

  public function handleErrorSend()
  {
    $this->forward('contact', 'index');
  }
}
```

Forms with array syntax

PHP allows you to use an array syntax for the form fields. When writing forms by yourslef, or when using the ones generated by the Propel admin, you end up with HTML code looking like:

```
<label for="story[title]">Titre:</label>
<input type="text" name="story[title]" id="story[title]" value="default value" size="45" />
```

The trouble is that using the input `id` as is (with brackets) in a validation file will push the YAML parser to its limits, and you will end up with errors. The solution here is to replace square brackets `[]` by curly brackets `{}` in the `names:` and `methods:` sections, and to use the explicit list declaration syntax with quotation marks for the `methods:` section:

```
methods:
  get:
    - "story{title}"
  post:
    - "story{title}"

names:
  story{title}:
    required:     Yes
```

Symfony will do the translation automatically, and the validation will run as expected.

Execute validator on an empty field

You sometimes need to execute a validator on a field which is not required, on an empty value. This happens, for instance, with a form where the user can (but may not) want to change his password, and in this case a confirmation password must be entered:

```
names:
  password1:
    required:     false
  password2:
    required:     false
    validators:   passwordValidator

passwordValidator:
   class:         sfCompareValidator
   param:
     check:       password1
     compare_error: The passwords you entered do not match. Please try again.
```

- If `password1 == null` and `password2 == null`
  - The `required` test passes
  - Validators are not run
  - The form is valid
- If `password2 == null` while `password1` is not null
  - The `required` test passes
  - Validators are not run
  - The form is valid

You may want to execute your `password2` validator IF `password1` is not null. Fortunately, the symfony validators handle this case, thanks to the `group` parameter. When a field is in a group, its validator will execute if it is not empty and if one of the fields of the same group is not empty.

So, if you change the configuration to:

```
names:
  password1:
    required:     false
    group:        password_group
  password2:
    required:     false
    group:        password_group
    validators:   passwordValidator

passwordValidator:
   class:         sfCompareValidator
   param:
     check:       password1
     compare_error: The passwords you entered do not match. Please try again.
```

Now, the validation occurs as follows:

- If `password1 == null` and `password2 == null`
  - The `required` test passes
  - Validators are not run
  - The form is valid
- If `password1 == null and password2 == 'foo'`
  - The `required` test passes
  - `password2` is not null, so its validator is executed - and it fails
  - An error message is thrown for `password2`
- If `password1 == 'foo' and password2 == null`
  - The `required` test passes
  - `password1` is not null, so the validator for `password2`, which is in the same group, is executed - and it fails
  - An error message is thrown for `password2`
- If `password1 == 'foo' and password2 == 'foo'`
  - The `required` test passes
  - `password2` is not null, so its validator is executed - and it passes
  - The form is valid

Client side validation

The client side validation is done by some javascript code. Symfony will soon provide built-in client-side validation features based on the YAML validation configuration files. Make sure you get the latest version !

How to repopulate a form from the request

Overview

The default value given to a form field usually depends either on the Model (if the form is displayed for the first time) and on the previous entry by the user (if the form is displayed for the second time). Handling these cases by hand can be cumbersome, especially since some form controls have a specific way of passing their value in the request parameters. That's why symfony handles the form repopulation (the process to fill in form controls from the request) with a special filter called the `sfFillInFormFilter`. There are two ways to use this filter, depending on whether the form has a validation or not.

Repopulation on a form without validation

The simplest example of a form that needs repopulation is a full-text search engine with a single text input.

```
<?php echo form_tag('product/find', 'method=get') ?>
  <?php echo input_tag('word', 'Enter a word') ?>
  <?php echo submit_tag('Search') ?>
</form>
```

When the user types a word and submits a query to the search engine (in this example, the `product/find` action), the form is usually displayed again in the result page, and the rules of user-friendly interfaces make that this form should display the word previously entered by the user. You could handle it manually by specifying a value for the input tag:

```
<?php echo input_tag('word', $sf_params->get('word', 'Enter a word')) ?>
```

But try to imagine more complicated cases, where the default value comes from the server, or where you have a radiobutton tag or a select tag. Handling all that becomes long and painful in the long run, especially if you have forms with a large number of controls.

Symfony can handle repopulation automatically. To enable that behaviour, you just need to add a `filters.yml` in your module `config/` folder and activate the `sfFillInFormFilter`:

```
myFillInFormFilter:
  class: sfFillInFormFilter
```

When displaying the form, symfony will check if the request contains parameters matching the form elements names and use them to fill the input tag automatically.

Repopulation of a form with validation

If you defined a validation file for a form (see how in the form validation chapter), you probably want the form to be displayed again with error messages and the value entered by the user when the validation fails. Instead of adding a `filter.yml` like above, you can directly declare the use of the `sfFillInFormFilter` in the validation file.

For instance, if a `product` form calls an `add` action, its validation uses a `validate/add.yml` file. To enable repopulation in the form on failed validation, add the following lines to the `add.yml` file:

```
fillin:
  activate: on
```

Filter parameters

The `sfFillInFormFilter` accepts additional parameters for the cases when you want to refine its action. The syntax is the same for a declaration in the `filters.yml` and in a validation file: just add a `param:` key to the filter declaration.

Page with multiple forms

If you want to restrict the action of the filter to a given form, for instance if your page contains more than one form, pass the name of the form to be repopulated as a parameter. For instance, a form named 'search':

```
// in the template
<?php echo form_tag('product/find', 'name=search') ?>
```

Is targeted in a `filter.yml` by:

```
myFillInFormFilter:
  class: sfFillInFormFilter
  param:
    name: search        # name of the form
```

If you prefer a validation file, the syntax is:

```
fillin:
  activate: on
  param:
    name: search        # name of the form
```

Converters

You may want to modify the data entered by the user before inserting and displaying it as values in the form. For instance, you may wish to remove any password from the request, to force the user to re-enter it (the same applies for captchas). Or you could desire to apply `htmlentities()` to the content of a textarea, to avoid scripting issues. This is all done through a `converters:` parameter, where you can specify a PHP function to be applied to one or more of the form inputs:

```
myFillInFormFilter:
  class: sfFillInFormFilter
  param:
    converters:
      htmlentities:     [word]
      serialize:        [site_url]
      empty_string:     [captcha, password]
```

As you can see, you can use existing PHP functions as converters or a function that you defined youself:

```
public function empty_string()
{
  return '';
}
```

How to send an email

Overview

Sending mails from a web application is a very common tasks. Symfony uses its architecture to automate it in a familiar way (MVC separation) and provides a specific class to deal with the particularities of the emails (multiple mime types, enclosed media, attachments).

Introduction

Symfony offers two ways to send emails from your web application:

- Via the `sfMail` class, which is a proxy class that offers an interface with PHPMailer. This solution is simple and fast, but doesn't provide MVC separation and is hardly compatible with i18n. Complex emails are also harder to compose with the `sfMail` class alone.
- Via a specific action and template. This solution is very versatile and deals with emails just as regular pages, with the addition of the specificities of this media. It is a little longer to put in place, but much more powerful than the first one.

The implementation of both solution will be illustrated through the same example: the sending of a forgotten password requested by a user.

Direct use of `sfMail`

The `sfMail` class will look familiar to those who know the `PHPMailer` class. It is simply a proxy class to `PHPMailer`, taking advantage of the symfony syntax. The `PHPMailer` class is included in the symfony package, so no additional installation (nor require) is required.

To send an email containing a password to a customer, an action has to do like the following:

```
public function executePasswordRequest()
{
  // determine customer from the request 'id' parameter
  $customer = CustomerPeer::retrieveByPk($this->getRequestParameter('id'));

  // class initialization
  $mail = new sfMail();
  $mail->initialize();
  $mail->setMailer('sendmail');
  $mail->setCharset('utf-8');

  // definition of the required parameters
  $mail->setSender('webmaster@my-company.com', 'My Company webmaster');
  $mail->setFrom('webmaster@my-company.com', 'My Company webmaster');
  $mail->addReplyTo('webmaster_copy@my-company.com');

  $mail->addAddress($customer->getEmail());

  $mail->setSubject('Your password request');
  $mail->setBody('
```

```
  Dear customer,

  You are so absentminded. Next time, try to remember your password:
  '.$customer->getPassword().'

  Regards,
  The My Company webmaster');

  // send the email
  $mail->send();
}
```

Use of an alternate action

In many cases, as the email sending process is just a detour in the logic of an action that does something else, it is often delegated to another action. Here is how it goes:

```
public function executePasswordRequest()
{
  // send the email
  $raw_email = $this->sendEmail('mail', 'sendPassword');

  // log the email
  $this->logMessage($raw_email, 'debug');
}
```

The email sending is delegated to a `sendPassword` action of a `mail` module. The `->sendEmail()` method of the `sfAction` class is a special kind of `->forward()` that executes another action but comes back afterward (it doesn't stop the execution of the current action). In addition, it returns a raw email that can be written into a log file (you will find more information about the way to log information in the debug chapter).

The `mail/sendPassword` deals with the `sfMail` object, but it doesn't need to define the mailer (it is taken from a configuration file) nor to initialize it:

```
public function executeSendPassword()
{
  // determine customer from the request 'id' parameter
  $customer = CustomerPeer::retrieveByPk($this->getRequestParameter('id'));

  // class initialization
  $mail = new sfMail();
  $mail->setCharset('utf-8');

  // definition of the required parameters
  $mail->setSender('webmaster@my-company.com', 'My Company webmaster');
  $mail->setFrom('webmaster@my-company.com', 'My Company webmaster');
  $mail->addReplyTo('webmaster_copy@my-company.com');

  $mail->addAddress($customer->getEmail());

  $mail->setSubject('Your password request');

  $this->password = $customer->getPassword();
  $this->mail = $mail;
```

```
}
```

Notice that the action doesn't need to call the `->send()` method for the `sfMail` object; being called by a `->sendEmail()`, it knows that it needs to finish by sending its `$this->mail` `sfMail` object. And where is the mail? will you ask. That's the beauty of the MVC separation: the body of the mail itself is to be written in the template `sendPasswordSuccess.php`:

```
Dear customer,

You are so absentminded. Next time, try to remember your password:
<?php echo $password ?>

Regards,
The My Company webmaster
```

> **Note**: If the `sendPassword` action ever determines that the email doesn't have to be sent, it can still abort the emailing process by returning `sfView::NONE`, just like a regular action.

Mailer configuration

If you use the alternate action method, you can (you don't have to) set the mailer and activate it environment by environment through a configuration file.

Create a `mailer.yml` configuration file in the `modules/mail/config/` directory with:

```
dev:
  deliver:    off

all:
  mailer:     sendmail
```

This stipulates the mailer program to be used to send mails, and deactivates the sending of mails in the development environment.

Send HTML email

Most of the time, emails are to be sent in HTML format, or even in a multipart format (enclosing both HTML and text format). To handle it directly with the `sfMail` object, use the `->setContentType()` method and specify an alternate body:

```
$mail->setContentType('text/html');
$mail->setAltBody('
<p>Dear customer</p>,
<p>
  You are so <i>absentminded</i>. Next time, try to remember your password:<br>
  <b>'.$customer->getPassword().'</b>
</p>
<p>
  Regards,<br>
  The My Company webmaster
</p>');
$mail->setAltBody('
```

```
Dear customer,

You are so absentminded. Next time, try to remember your password:
'.$customer->getPassword().'

Regards,
The My Company webmaster');
```

If you use an alternate action, you will just need to use an alternate template ending with `.altbody.php`. Symfony will automatically add it as alternate body:

```
// in sendPasswordSuccess.php
<p>Dear customer</p>,
<p>
  You are so <i>absentminded</i>. Next time, try to remember your password:<br>
  <b><?php echo $password ?></b>
</p>
<p>
  Regards,<br>
  The My Company webmaster
</p>

// in sendPasswordSuccess.altbody.php
Dear customer,

You are so absentminded. Next time, try to remember your password:
<?php echo $password ?>

Regards,
The My Company webmaster
```

> **Note**: If you just use an HTML version without altbody template, you will need to set the content type to `text/html` in the `sendPassword` action.

Embed images

HTML emails can contain images directly embedded in the body. To add an embedded image, use the `->addEmbeddedImage()` method of the `sfMail` object:

```
$mail->addEmbeddedImage(sfConfig::get('sf_web_dir').'/images/my_company_logo.gif', 'CID1', 'My Co
```

The first argument is the path to the image, the second is a reference of the image that you can use in the template to embed it:

```
// in sendPasswordSuccess.php
<p>Dear customer</p>,
<p>
  You are so <i>absentminded</i>. Next time, try to remember your password:<br>
  <b><?php echo $password ?></b>
</p>
<p>
  Regards,<br>
  The My Company webmaster
  <img src="cid:CID1" />
</p>
```

Attachments

Attaching a document to a mail is as simple as you would expect it to be:

```
// document attachment
$mail->addAttachment(sfConfig::get('sf_data_dir').'/MyDocument.doc');
// string attachment
$mail->addStringAttachment('this is some cool text to embed', 'file.txt');
```

Email addresses advanced syntax

In addition to recipients, emails often need to be sent as carbon copy ('cc:') or blind carbon copy ('bcc:'). Here is how to do this with `sfMail`:

```
$mail->addAddress($customer->getEmail());
$mail->addCc('carbon_copy@my-companyt.com ');
$mail->addBcc('blind_carbon_copy@my-company.com');
```

The `sfMail` methods used to set emails (`->setSender()`, `->setFrom()`, `->addReplyTo()`, `->addAddress()`, `->addCc()`, `->addBcc()`) can use two syntaxes:

```
$mail->setFrom('me@symfony-project.com', 'Symfony');
// is equivalent to
$mail->setFrom('me@symfony-project.com <Symfony>');
```

In addition, to minimize the code in case of multiple recipients, `sfMail` has an `->addAddresses()` method:

```
$mail->addAddress('client1@client.com <Jules>');
$mail->addAddress('client2@client.com <Jim>');
// is equivalent to
$mail->addAddresses(array('client1@client.com <Jules>', 'client2@client.com <Jim>'));
```

`sfMail` methods

Once you've built your `sfMail` object, you might want to check its content. Fortunately, all the setter methods described above have an equivalent getter method, and you can clear the properties previously set:

```
->setCharset($charset)
->getCharset()
->setContentType($content_type)
->getContentType()
->setPriority($priority)
->getPriority()
->setEncoding($encoding)
->getEncoding()
->setSubject($subject)
->getSubject()
->setBody($body)
->getBody()
->setAltBody($text)
->getAltBody()
```

```
->setMailer($type = 'mail')
->getMailer()
->setSender($address, $name = null)
->getSender()
->setFrom($address, $name = null)
->getFrom()
->addAddresses($addresses)
->addAddress($address, $name = null)
->addCc($address, $name = null)
->addBcc($address, $name = null)
->addReplyTo($address, $name = null)
->clearAddresses()
->clearCcs()
->clearBccs()
->clearReplyTos()
->clearAllRecipients()
->addAttachment($path, $name = '', $encoding = 'base64', $type = 'application/octet-stream')
->addStringAttachment($string, $filename, $encoding = 'base64', $type = 'application/octet-stream
->addEmbeddedImage($path, $cid, $name = '', $encoding = 'base64', $type = 'application/octet-stre
->setAttachments($attachments)
->clearAttachments()
->addCustomHeader($name, $value)
->clearCustomHeaders()
->setWordWrap($wordWrap)
->getWordWrap()
```

How to achieve persistent sessions with cookies?

Overview

Symfony offers access to cookies via the `sfWebRequest` and `sfWebResponse` objects. It makes the use of cookies very easy, and persistent sessions are easily achieved.

Cookie getter and setter

A cookie is a string stored on the client's computer, written by a web application and readable only by the same application - or domain.

In symfony, the setter and getter for cookies are methods of different objects, but that makes sense. To get a cookie, you inspect the request that was sent to the server, thus using the `sfWebRequest` object. On the other hand, to set a cookie, you modify the response that will be sent to the user, thus using the `sfWebResponse` object. To manipulate cookies from within an action, use the following shorcuts:

```
// cookie getter
$string = $this->getRequest()->getCookie('mycookie');

// cookie setter
$this->getResponse()->setCookie('mycookie', $value);

// cookie setter with options
$this->getResponse()->setCookie('mycookie', $value, $expire, $path, $domain, $secure);
```

The syntax of the `->setCookie()` method is the same as the one of the basic PHP `setcookie()` function (refer to the PHP manual for more information). The main advantage of using the `sfWebResponse` method is that symfony logs cookies, and that you can keep on reading and modifying them until the response is actually sent.

> **Note**: If you want to manipulate cookies outside of an action, you will need to access the `Request` and `Answer` objects without shortcut:
>
> ```
> $request  = sfContext::getInstance()->getRequest();
> $response = sfContext::getInstance()->getResponse();
> ```

Persistent sessions

A good use for cookies (apart from basic session handling, which is completely transparent in symfony) is the persistent sessions functionality. Most of the login forms offer a "remember me" check-box which, when clicked, allows the user to bypass the login process for future sessions.

Basic login

Let's imagine an application where all the modules are secure except the `security` module. The `settings.yml` is configured to handle the request of unauthenticated users to the `security/index` action:

```
all:
  .settings:
    login_module:           security
    login_action:           index
```

The model has a `User` class with at the very least a `login` and a `password` field. The `indexSuccess.php` template shows a login form (without the "remember me" checkbox for now), and handles the submission to the `security/login` action:

```
public function executeIndex()
{
}

public function executeLogin()
{
  // check if the user exists
  $c = new Criteria();
  $c->add(UserPeer::LOGIN, $this->getRequestParameter('login'));
  $user = UserPeer::doSelectOne($c);
  if ($user)
  {
    // check if the password is correct
    if ($this->getRequestParameter('password') == $user->getPassword())
    {
      // sign in
      $this->getContext()->getUser()->signIn();
      // proceed to home page
      return $this->redirect('main/index');
    }
    else
    {
      $this->getRequest()->setError('password', 'wrong password');
    }
  }
  else
  {
    $this->getRequest()->setError('email', 'this user does not exist');
  }

  // an error was found
  return $this->forward('security', 'index');
}
```

> **Note**: The verification of the login and password could also be handled in a custom validator for a better domain model logic, as explained in the askeet tutorial.

Now, let's have a look at this `->signIn()` method in the `myUser` class:

```
class myUser extends sfBasicSecurityUser
{
  public function signIn()
  {
    $this->setAuthenticated(true);
  }

  public function signOut()
  {
```

```
    $this->setAuthenticated(false);
  }
}
```

So far this is very basic. But it works fine, as long as you ask the user to login for every session.

Persistent sessions

To allow for persistent sessions, the server has to store some information in the client's computer (that is were the cookie comes in) remembering who the user is and that he/she successfully logged in before. Of course, for security reasons, the password cannot be stored in the cookie (and, by the way, that would be incompatible with the sha1 hash password storage method described in the askeet tutorial). So what should be stored in the cookie then? Whatever the cookie stores, it has to be the some data that can be matched to what is in the database, so that the comparison of the two elements achieves the authentication. So, to minimize the risk, a random string will be stored and regenerated every 15 days (the lifetime that will be given to the cookie).

By adding a new `remember_key` column to the `User` table (and rebuilding the model). This new field will store the random key, the key will thus be stored both in the cookie on the client's computer and in the database as part of the user's record. The remember key will be set when a user requests to be remembered, so change the sign-in line in the `login` action by:

```
// sign in
$remember = $this->getRequestParameter('remember_me');
$this->getContext()->getUser()->signIn($user, $remember);
```

Don't forget to add a `remember_me` checkbox to the `modules/security/templates/indexSuccess.php` form for this to work.

The `->signIn()` method of the `myUser` class has to be modified to set the remember key both in the database and in the cookie:

```
public function signIn($user, $remember = false)
{
  $this->setAuthenticated(true);

  if ($remember)
  {
    // determine a random key
    if (!$user->getRememberKey())
    {
      $rememberKey = myTools::generate_random_key();

      // save the key to the User table
      $user->setRememberKey($rememberKey);
      $user->save();
    }

    // save the key to the cookie
    $value = base64_encode(serialize(array($user->getRememberKey(), $user->getLogin())));
    sfContext::getInstance()->getResponse()->setCookie('MyWebSite', $value, time()+60*60*24*15, '
  }
}
```

The `generate_random_key()` method can be anything that you choose which meets with your security requirements. Now, you just need to change the `security/index` action a little bit:

```
public function executeIndex()
{
  if ($this->getRequest()->getCookie('MyWebSite'))
  {
    $value = unserialize(base64_decode($this->getRequest()->getCookie('MyWebSite')));
    $c = new Criteria();
    $c->add(UserPeer::REMEMBER_KEY, $value[0]);
    $c->add(UserPeer::LOGIN, $value[1]);
    $user = UserPeer::doSelectOne($c);
    if ($user)
    {
      // sign in
      $this->getContext()->getUser()->signIn($user);
      // proceed to home page
      return $this->redirect('main/index');
    }
  }
}
```

This new process reads the cookie and you are done.

> **Note**: If some pages of your website are accessible without authentication, then the `security/index` action is no longer the first action to be executed every time. In order to automatically log users in such cases, you will probably prefer to add a new `rememberFilter` in your application `lib/` directory instead of doing the cookie check in a single action:
>
> ```
> class rememberFilter extends sfFilter
> {
>   public function execute ($filterChain)
>   {
>     // execute this filter only once
>     if ($this->isFirstCall())
>     {
>       if ($cookie = $this->getContext()->getRequest()->getCookie('MyWebSite'))
>       {
>         $value = unserialize(base64_decode($cookie));
>         $c = new Criteria();
>         $c->add(UserPeer::REMEMBER_KEY, $value[0]);
>         $c->add(UserPeer::LOGIN, $value[1]);
>         $user = UserPeer::doSelectOne($c);
>         if ($user)
>         {
>           // sign in
>           $this->getContext()->getUser()->signIn($user);
>         }
>       }
>     }
>     // execute next filter
>     $filterChain->execute();
>   }
> }
> ```

Of course, you will have to declare this filter in your application `filters.yml` configuration file:

```
rememberFilter:
   class: rememberFilter
```

One last thing: If the user logs out, don't forget to remove the cookie!

```
public function signOut()
{
  $this->setAuthenticated(false);
  sfContext::getInstance()->getResponse()->setCookie('MyWebSite', '', time() - 3600, '/');
}
```

Note: This solution works only for non-secure pages, since the security check for secure pages occurs *before* the custom `remember` filter. Consequently, if a user with a proper cookie tries to access a secure page without having logged to the site in a non-secure page before, he/she will be redirected to the login page as anybody else. If you want the remember me feature to work for secure pages as well, the implementation has to be slightly different. You must create a `myBasicSecurityFilter` class, specializing the `sfBasicSecurityFilter` class, and put the cookie control in it. Then, in the `factories.yml`, change the name of the `security_filter` class to this `myBasicSecurityFilter`. The details of the implementation are left to your sagacity.

How to display a custom 404 error page

Overview

With symfony, you can easily define a custom error 404 page. You even have several ways to do it.

Introduction

The module and action to be called when a 404 error occurs is defined in the `settings.yml` configuration file of each application:

```
all:
  .actions:
    error_404_module:       default
    error_404_action:       error404
```

The `error404` action of the `default` module doesn't appear in the application directory, since it is part of symfony.

Solution 1 : change the configuration

You can specify a custom module and action in the `settings.yml` file:

```
default:
  .actions:
    error_404_module:       errors
    error_404_action:       error404
```

Just initialize the new module with:

```
$ symfony init-module myapp errors
```

You need to create the action `error404` in the file `myproject/apps/myapp/modules/errors/actions/action.class.php`, and the template `error404Success.php` in the directory `myproject/apps/myapp/modules/errors/templates/`. You now have full control over the content of your 404 error page.

Solution 2 : override the default template

A simpler way to do it is to override the default `error404Success.php` template. It must be located in the `default` module, so create it if it doesn't exist already.

```
$ cd myapp/modules
$ mkdir default
```

Symfony will first look for an action or a template here before calling the default ones. So create a file called `error404Success.php` in a newly created `templates` directory, write whatever you need into it, and

you're done.

```
$ cd default
$ mkdir templates
$ cd templates
$ vi "error404Success.php"
```

> **Note**: Why not use the `symfony init-module` here? Because it would create an `actions.class.php` file that would override all the actions of the default `default` module, including `index`, `login`, `disabled`, `secure` and `unavailable`. So using the `symfony init-module` command would imply *erasing* the newly created `actions.class.php` file to focus only on the template.