

symfony

A Gentle Introduction to symfony

Build professional websites faster with PHP and symfony

symfony 1.3 & 1.4

This PDF is brought to you by
SENSIO LABS 

License: GFDL

Version: gentle-introduction-1.4-en-2012-08-29

Table of Contents

Chapter 1: Introducing Symfony.....	11
Symfony in Brief	11
Symfony Features	11
Who Made Symfony and Why?	12
The Symfony Community.....	13
Is Symfony for Me?.....	14
Fundamental Concepts	14
PHP	14
Object-Oriented Programming (OOP)	14
Magic Methods	15
Object-Relational Mapping (ORM)	15
Rapid Application Development (RAD).....	16
YAML	17
Summary.....	18
Chapter 2: Exploring Symfony's Code.....	19
The MVC Pattern	19
MVC Layering.....	20
Layer Separation Beyond MVC	23
Symfony's MVC Implementation	26
Symfony Core Classes	28
Code Organization	29
Project Structure: Applications, Modules, and Actions	29
File Tree Structure	29
Common Instruments	33
Parameter Holders	33
Constants.....	35
Class Autoloading	35
Summary.....	36
Chapter 3: Running Symfony.....	37
Prerequisites.....	37
Third-Party Software	37
Command Line Interface	37
PHP Configuration.....	38
Symfony Installation	38
Initializing the Project Directory	38
Choosing the Symfony Version.....	39
Choosing the Symfony Installation Location	39
Installing Symfony	39
Project Setup	41
Project Creation	41
Configuring the Database.....	41
Application Creation.....	42

Directory Structure Rights	42
Web Server Configuration	43
The ugly Way	43
The secure Way	43
Using the Sandbox.....	46
Summary.....	47
Chapter 4: The Basics Of Page Creation.....	48
Creating a Module Skeleton	48
Adding a Page.....	50
Adding an Action	50
Adding a Template.....	51
Passing Information from the Action to the Template	52
Linking to Another Action.....	53
Getting Information from the Request	54
Summary.....	55
Chapter 5: Configuring Symfony	56
The Configuration System	56
YAML Syntax and Symfony Conventions.....	57
Help, a YAML File Killed My App!	60
Overview of the Configuration Files	61
Project Configuration	61
Application Configuration	61
Module Configuration.....	63
Environments.....	64
What Is an Environment?	64
Configuration Cascade	66
The Configuration Cache	68
Accessing the Configuration from Code	69
The <code>sfConfig</code> Class	69
Custom Application Settings and <code>app.yml</code>	70
Tips for Getting More from Configuration Files	71
Using Constants in YAML Configuration Files	71
Using Scriptable Configuration.....	72
Browsing Your Own YAML File	72
Summary.....	73
Chapter 6: Inside The Controller Layer.....	74
The Front Controller.....	74
The Front Controller's Job in Detail	74
The Default Front Controller.....	75
Calling Another Front Controller to Switch the Environment	75
Actions	76
The Action Class	76
Alternative Action Class Syntax	77
Retrieving Information in the Action.....	78
Action Termination	79
Skipping to Another Action	81
Repeating Code for Several Actions of a Module	82
Accessing the Request.....	83
User Session	84
Accessing the User Session.....	84
Flash Attributes	86
Session Management.....	86

Action Security	88
Access Restriction	88
Granting Access.....	89
Complex Credentials	91
Filters.....	91
The Filter Chain.....	91
Building Your Own Filter.....	93
Filter Activation and Parameters	94
Sample Filters	95
Module Configuration.....	96
Summary.....	97
Chapter 7: Inside The View Layer	98
Templating	98
Helpers	99
Page Layout.....	101
Template Shortcuts	103
Code Fragments	103
Partials	103
Components.....	105
Slots.....	107
View Configuration	110
The <code>view.yml</code> File.....	110
The Response Object	111
View Configuration Settings.....	112
Output Escaping	117
Activating Output Escaping.....	118
Escaping Helpers	119
Escaping Arrays and Objects.....	119
Summary	120
Chapter 8: Inside The Model Layer (Doctrine)	121
Why Use an ORM and an Abstraction Layer?.....	121
Symfony's Database Schema	122
Schema Example	123
Basic Schema Syntax.....	124
Model Classes	124
Base and Custom Classes	125
Object and Table Classes	125
Accessing Data.....	126
Retrieving the Column Value	126
Retrieving Related Records.....	126
Saving and Deleting Data.....	127
Retrieving Records by Primary Key	128
Retrieving Records with Doctrine_Query.....	128
Using Raw SQL Queries	130
Using Special Date Columns	131
Database Connections	132
Extending the Model.....	133
Adding New Methods	134
Overriding Existing Methods	134
Using Model Behaviors.....	135
Extended Schema Syntax	135
Attributes.....	135
Column Details	138

Relationships	138
Indexes	139
I18n Tables	139
Behaviors	139
Don't Create the Model Twice	140
Building a SQL Database Structure Based on an Existing Schema	140
Generating a YAML Data Model from an Existing Database	140
Summary	141
Chapter 9: Links And The Routing System	142
What Is Routing?	142
URLs As Server Instructions	142
URLs As Part of the Interface.....	143
How It Works.....	144
URL Rewriting	146
Link Helpers	147
Hyperlinks, Buttons, and Forms.....	148
Link Helper Options	149
Fake GET and POST Options.....	149
Forcing Request Parameters As GET Variables	150
Using Absolute Paths.....	151
Routing Configuration	152
Rules and Patterns	152
Pattern Constraints	154
Setting Default Values.....	155
Speeding Up Routing by Using the Rule Name	156
Creating Rules Without <code>routing.yml</code>	157
Dealing with Routes in Actions.....	157
Summary.....	158
Chapter 10: Forms	159
Displaying a Form.....	159
Customizing the Form Display.....	161
Form Widgets	163
Standard Widgets	164
List Widgets.....	164
Foreign Key Widgets	166
Date Widgets	167
I18n Widgets	168
File Widgets.....	169
Handling a Form Submission	170
Simple Form Handling	170
Form Handling With Data Validation	171
Using Clean Form Data	172
Customizing Error Messages Display.....	174
Customizing Validators	174
Applying a Validator To Several Fields	175
Validators.....	176
Alternative Ways to Use a Form	178
Form Classes	178
Altering a Form Object.....	180
Custom Widget and Validator classes	180
Forms Based on a Model	183
Generating Model Forms.....	183
Using Model Forms	185

Conclusion	186
Chapter 11: Emails.....	187
Introduction	187
Sending Emails from an Action	187
The Fastest Way	187
The Flexible Way	188
The Powerful Way.....	188
Using the Symfony View.....	189
Configuration.....	189
The Delivery Strategy	190
The realtime Strategy	190
The single_address Strategy	190
The spool Strategy	190
The none Strategy.....	192
The Mail Transport	192
Sending an Email from a Task.....	193
Debugging	193
Testing	194
Email Messages as Classes.....	195
Recipes.....	197
Sending Emails via Gmail.....	197
Customizing the Mailer Object.....	197
Using Swift Mailer Plugins	197
Customizing the Spool Behavior	198
Chapter 12: Caching	201
Caching the Response	201
Global Cache Settings	201
Caching an Action	202
Caching a Partial or Component	203
Caching a Template Fragment.....	205
Configuring the Cache Dynamically	206
Using the Super Fast Cache.....	208
Removing Items from the Cache	209
Clearing the Entire Cache	209
Clearing Selective Parts of the Cache	210
Clearing several cache parts at once	212
Clearing cache across applications	213
Testing and Monitoring Caching	213
Building a Staging Environment	214
Monitoring Performance	214
Benchmarking	215
Identifying Cache Parts	215
HTTP 1.1 and Client-Side Caching	215
Adding an ETag Header to Avoid Sending Unchanged Content	215
Adding a Last-Modified Header to Avoid Sending Still Valid Content	216
Adding Vary Headers to Allow Several Cached Versions of a Page	216
Adding a Cache-Control Header to Allow Client-Side Caching	217
Summary	217
Chapter 13: I18n And L10n.....	219
User Culture	219
Setting the Default Culture	219
Changing the Culture for a User	220

Determining the Culture Automatically	221
Standards and Formats	222
Outputting Data in the User's Culture	222
Getting Data from a Localized Input	223
Text Information in the Database	223
Creating Localized Schema	224
Using the Generated I18n Objects	225
Interface Translation	225
Configuring Translation	225
Using the Translation Helper	226
Using Dictionary Files	226
Managing Dictionaries	227
Handling Other Elements Requiring Translation	228
Handling Complex Translation Needs	228
Calling the Translation Helper Outside a Template	230
Summary	230
Chapter 14: Admin Generator.....	231
Code Generation Based on the Model	231
Example Data Model	231
Administration	233
Initiating an Administration Module	233
A Look at the Generated Code	235
Introducing the <code>generator.yml</code> Configuration File	236
Generator Configuration	238
Fields	238
View Customization	243
List View-Specific Customization	245
New and Edit View-Specific Customization	250
Dealing with Foreign Keys	251
Adding Interactions	252
Form Validation	254
Restricting User Actions Using Credentials	254
Modifying the Presentation of Generated Modules	255
Using a Custom Style Sheet	255
Creating a Custom Header and Footer	255
Customizing the Theme	256
Summary	258
Chapter 15: Unit And Functional Testing.....	259
Automated Tests	259
Unit and Functional Tests	259
Test-Driven Development	260
The Lime Testing Framework	260
Unit Tests	261
What Do Unit Tests Look Like?	261
Unit Testing Methods	262
Testing Parameters	264
The <code>test:unit</code> Task	265
Stubs, Fixtures, and Autoloading	265
Unit testing ORM classes	268
Functional Tests	269
What Do Functional Tests Look Like?	269
Browsing with the <code>sfBrowser</code> Object	270
Using Assertions	272

Using CSS Selectors	274
Testing for errors	275
Working in the Test Environment	276
The <code>test:functional</code> Task.....	276
Test Naming Practices.....	277
Special Testing Needs	278
Executing Tests in a Test Harness	278
Accessing a Database	279
Testing the Cache.....	280
Testing Interactions on the Client.....	280
Summary.....	282
Chapter 16: Application Management Tools	283
Logging	283
PHP Logs	283
Symfony Logs	284
Debugging	287
Symfony Debug Mode	287
Symfony Exceptions	288
Xdebug Extension.....	288
Web Debug Toolbar.....	289
Manual Debugging	293
Using symfony outside of a web context	294
Batch Files.....	294
Custom Tasks	295
Populating a Database	297
Fixture File Syntax	297
Launching the Import.....	298
Using Linked Tables	298
Deploying Applications	299
Using <code>rsync</code> for Incremental File Transfer	299
Ignoring Irrelevant Files	301
Managing a Production Application	302
Summary.....	303
Chapter 17: Extending Symfony	304
Events	304
Understanding Events	304
Notifying an Event listener.....	305
Notifying the dispatcher of an Event Until a Listener handles it.....	306
Changing the Return Value of a Method.....	308
Built-In Events	309
Where To Register Listeners?	311
Factories	311
Plug-Ins	312
Finding Symfony Plug-Ins.....	313
Installing a Plug-In	313
Anatomy of a Plug-In	316
Doctrine	321
Propel	321
How to Write a Plug-In	322
Summary.....	327
Chapter 18: Performance.....	328
Tweaking the Server.....	328

Tweaking the Model	329
Optimizing Propel or Doctrine Integration	329
Limiting the Number of Objects to Hydrate.....	330
Minimizing the Number of Queries with Joins	330
Avoid Using Temporary Arrays	333
Bypassing the ORM	334
Speeding Up the Database	335
Tweaking the View.....	336
Using the Fastest Code Fragment.....	336
Speeding Up the Routing Process	337
Skipping the Template	337
Tweaking the Cache	338
Clearing Selective Parts of the Cache	338
Generating Cached Pages	339
Using a Database Storage System for Caching.....	340
Bypassing Symfony.....	340
Caching the Result of a Function Call.....	340
Caching Data in the Server	341
Deactivating the Unused Features	341
Optimizing Your Code.....	343
Core Compilation.....	343
The project:optimize Task.....	343
Summary.....	343
Chapter 19: Mastering Symfony's Configuration Files	344
Symfony Settings	344
Default Modules and Actions.....	344
Optional Feature Activation	346
Feature Configuration	347
Extending the Autoloading Feature.....	350
Custom File Structure	352
The Basic File Structure.....	352
Customizing the File Structure	353
Modifying the Project Web Root	353
Understanding Configuration Handlers	354
Default Configuration Handlers	354
Adding Your Own Handler.....	355
Summary.....	357
Appendix A: Inside The Model Layer (Propel)	359
Why Use an ORM and an Abstraction Layer?.....	359
Symfony's Database Schema	360
Schema Example	361
Basic Schema Syntax.....	361
Model Classes	362
Base and Custom Classes	363
Object and Peer Classes	363
Accessing Data.....	364
Retrieving the Column Value	364
Retrieving Related Records.....	365
Saving and Deleting Data.....	366
Retrieving Records by Primary Key	367
Retrieving Records with Criteria.....	367
Using Raw SQL Queries	370
Using Special Date Columns	370

Database Connections	371
Extending the Model.....	373
Adding New Methods	373
Overriding Existing Methods	374
Using Model Behaviors.....	374
Extended Schema Syntax	375
Attributes.....	375
Column Details	377
Foreign Keys.....	378
Indexes	379
Empty Columns	379
I18n Tables	380
Behaviors.....	380
Beyond the schema.yml: The schema.xml.....	381
Don't Create the Model Twice	382
Building a SQL Database Structure Based on an Existing Schema	382
Generating a YAML Data Model from an Existing Database	382
Summary.....	383
Appendix B: GNU Free Documentation License	385
0. PREAMBLE	385
1. APPLICABILITY AND DEFINITIONS.....	385
2. VERBATIM COPYING	387
3. COPYING IN QUANTITY	387
4. MODIFICATIONS	387
5. COMBINING DOCUMENTS	389
6. COLLECTIONS OF DOCUMENTS	389
7. AGGREGATION WITH INDEPENDENT WORKS	389
8. TRANSLATION	390
9. TERMINATION	390
10. FUTURE REVISIONS OF THIS LICENSE	390

Chapter 1

Introducing Symfony

What can symfony do for you? What's required to use it? This chapter answers these questions.

Symfony in Brief

A framework streamlines application development by automating many of the patterns employed for a given purpose. A framework also adds structure to the code, prompting the developer to write better, more readable, and more maintainable code. Ultimately, a framework makes programming easier, since it packages complex operations into simple statements.

Symfony is a complete framework designed to optimize the development of web applications by way of several key features. For starters, it separates a web application's business rules, server logic, and presentation views. It contains numerous tools and classes aimed at shortening the development time of a complex web application. Additionally, it automates common tasks so that the developer can focus entirely on the specifics of an application. The end result of these advantages means there is no need to reinvent the wheel every time a new web application is built!

Symfony is written entirely in PHP. It has been thoroughly tested in various real¹-world² projects³, and is actually in use for high-demand e-business websites. It is compatible with most of the available databases engines, including MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. It runs on *nix and Windows platforms. Let's begin with a closer look at its features.

Symfony Features

Symfony was built in order to fulfill the following requirements:

- Easy to install and configure on most platforms (and guaranteed to work on standard *nix and Windows platforms)
- Database engine-independent
- Simple to use, in most cases, but still flexible enough to adapt to complex cases
- Based on the premise of convention over configuration—the developer needs to configure only the unconventional
- Compliant with most web best practices and design patterns

1. <http://sf-to.org/answers>
2. <http://sf-to.org/delicious>
3. <http://sf-to.org/dailymotion>

- Enterprise-ready—adaptable to existing information technology (IT) policies and architectures, and stable enough for long-term projects
- Very readable code, with `phpDocumentor` comments, for easy maintenance
- Easy to extend, allowing for integration with other vendor libraries

Automated Web Project Features

Most of the common features of web projects are automated within `symfony`, as follows:

- The built-in internationalization layer allows for both data and interface translation, as well as content localization.
- The presentation uses templates and layouts that can be built by HTML designers without any knowledge of the framework. Helpers reduce the amount of presentation code to write by encapsulating large portions of code in simple function calls.
- Forms support automated validation and repopulation, and this ensures a good quality of data in the database and a better user experience.
- Output escaping protects applications from attacks via corrupted data.
- The cache management features reduce bandwidth usage and server load.
- Authentication and credential features facilitate the creation of restricted sections and user security management.
- Routing and smart URLs make the page address part of the interface and search-engine friendly.
- Built-in e-mail and API management features allow web applications to go beyond the classic browser interactions.
- Lists are more user-friendly thanks to automated pagination, sorting, and filtering.
- Factories, plug-ins, and events provide a high level of extensibility.

Development Environment and Tools

To fulfill the requirements of enterprises having their own coding guidelines and project management rules, `symfony` can be entirely customized. It provides, by default, several development environments and is bundled with multiple tools that automate common software-engineering tasks:

- The code-generation tools are great for prototyping and one-click back-end administration.
- The built-in unit and functional testing framework provides the perfect tools to allow test-driven development.
- The debug panel accelerates debugging by displaying all the information the developer needs on the page he's working on.
- The command-line interface automates application deployment between two servers.
- Live configuration changes are possible and effective.
- The logging features give administrators full details about an application's activities.

Who Made Symfony and Why?

The first version of `symfony` was released in October 2005 by project founder Fabien Potencier, coauthor of this book. Fabien is the CEO of Sensio (<http://www.sensio.com/>⁴), a French web agency well known for its innovative views on web development.

Back in 2003, Fabien spent some time inquiring about the existing open source development tools for web applications in PHP. He found that none fulfilled the previously described requirements. When PHP 5 was released, he decided that the available tools had reached a

4. <http://www.sensio.com/>

mature enough stage to be integrated into a full-featured framework. He subsequently spent a year developing the symfony core, basing his work on the Mojavi Model-View-Controller (MVC) framework, the Propel object-relational mapping (ORM), and the Ruby on Rails templating helpers.

Fabien originally built symfony for Sensio's projects, because having an effective framework at your disposal presents an ideal way to develop applications faster and more efficiently. It also makes web development more intuitive, and the resulting applications are more robust and easier to maintain. The framework entered the proving grounds when it was employed to build an e-commerce website for a lingerie retailer, and subsequently was applied to other projects.

After successfully using symfony for a few projects, Fabien decided to release it under an open source license. He did so to donate this work to the community, to benefit from user feedback, to showcase Sensio's experience, and because it's fun.



Why "symfony" and not "FooBarFramework"? Because Fabien wanted a short name containing an s, as in Sensio, and an f, as in framework—easy to remember and not associated with another development tool. Also, he doesn't like capital letters. symfony was close enough, even if not completely English, and it was also available as a project name. The other alternative was "baguette".

For symfony to be a successful open source project, it needed to have extensive documentation, in English, to increase the adoption rate. Fabien asked fellow Sensio employee François Zaninotto, the other author of this book, to dig into the code and write an online book about it. It took quite a while, but when the project was made public, it was documented well enough to appeal to numerous developers. The rest is history.

The Symfony Community

As soon as the symfony website (<http://www.symfony-project.org/>⁵) was launched, numerous developers from around the world downloaded and installed the framework, read the online documentation, and built their first application with symfony, and the buzz began to mount.

Web application frameworks were getting popular at that time, and the need for a full-featured framework in PHP was high. Symfony offered a compelling solution due to its impressive code quality and significant amount of documentation—two major advantages over the other players in the framework category. Contributors soon began to surface, proposing patches and enhancements, proofreading the documentation, and performing other much-needed roles.

The public source repository and ticketing system offer a variety of ways to contribute, and all volunteers are welcome. Fabien is still the main committer in the trunk of the source code repository, and guarantees the quality of the code.

Today, the symfony forum⁶, mailing⁷ lists⁸, and Internet Relay Chat (IRC) channel⁹ offer ideal support outlets, with seemingly each question getting an average of four answers. Newcomers install symfony every day, and the wiki and code snippets sections host a lot of user-contributed documentation. Nowadays, symfony is one of the most popular PHP frameworks.

The symfony community is the third strength of the framework, and we hope that you will join it after reading this book.

5. <http://www.symfony-project.org/>

6. <http://forum.symfony-project.org/>

7. <http://groups.google.com/group/symfony-users>

8. <http://groups.google.com/group/symfony-devs>

9. <irc://irc.freenode.net/symfony>

Is Symfony for Me?

Whether you are a PHP expert or a newcomer to web application programming, you will be able to use symfony. The main factor in deciding whether or not to do so is the size of your project.

If you want to develop a simple website with five to ten pages, limited access to a database, and no obligations to ensuring its performance or providing documentation, then you should stick with PHP alone. You wouldn't gain much from a web application framework, and using object orientation or an MVC model would likely only slow down your development process. As a side note, symfony is not optimized to run efficiently on a shared server where PHP scripts can run only in Common Gateway Interface (CGI) mode.

On the other hand, if you develop more complex web applications, with heavy business logic, PHP alone is not enough. If you plan on maintaining or extending 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. In all these cases, symfony is 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 that is mature, well documented, and has a large community. Search no more, for symfony is your solution.



If you would like a visual demonstration, take a look at the screencasts available from the symfony website. You will see how fast and fun it is to develop applications with symfony.

Fundamental Concepts

Before you get started with symfony, you should understand a few basic concepts. Feel free to skip ahead if you already know the meaning of OOP, ORM, RAD, DRY, KISS, TDD, and YAML.

PHP

Symfony is developed in PHP (<http://www.php.net/>¹⁰) and dedicated to building web applications with the same language. Therefore, a solid understanding of PHP and object-oriented programming is required to get the most out of the framework. The minimal version of PHP required to run symfony is PHP 5.2.4.

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) will not be explained in this chapter. It needs a whole book itself! Because symfony makes extensive use of the object-oriented mechanisms available as of PHP 5, OOP is a prerequisite to learning symfony.

Wikipedia explains OOP as follows:

"The idea behind object-oriented programming is that a computer program may be seen as comprising a collection of individual units, or objects, that act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer."

PHP implements the object-oriented paradigms of class, object, method, inheritance, and much more. Those who are not familiar with these concepts are advised to read the related PHP documentation, available at <http://www.php.net/manual/en/language.oop5.basic.php>¹¹.

10. <http://www.php.net/>

Magic Methods

One of the strengths of PHP's object capabilities is the use of magic methods. These are methods that can be used to override the default behavior of classes without modifying the outside code. They make the PHP syntax less verbose and more extensible. They are easy to recognize, because the names of the magic methods start with two underscores (`__`).

For instance, when displaying an object, PHP implicitly looks for a `__toString()` method for this object to see if a custom display format was defined by the developer:

```
$myObject = new myClass();
echo $myObject;

// Will look for a magic method
echo $myObject->__toString();
```

Listing 1-1

Symfony uses magic methods, so you should have a thorough understanding of them. They are described in the PHP documentation (<http://www.php.net/manual/en/language.oop5.magic.php>¹²).

Object-Relational Mapping (ORM)

Databases are relational. PHP 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. This interface is called an object-relational mapping, or ORM.

An ORM 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 has not 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 decide. Just change one line in a configuration file, and it works.

An abstraction layer encapsulates the data logic. The rest of the application does not need to know about the SQL queries, and the SQL that accesses the database is easy to find. Developers who specialize in database programming also know clearly where to go.

Using objects instead of records, and classes instead of tables, has another benefit: you can 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, this is as easy as adding a new accessor method to the `Client` class, like this:

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

Listing 1-2

All the repeated data-access functions and the business logic of the data can be maintained within such objects. For instance, consider a class `ShoppingCart` in which you keep items

11. <http://www.php.net/manual/en/language.oop5.basic.php>

12. <http://www.php.net/manual/en/language.oop5.magic.php>

(which are objects). To retrieve the full amount of the shopping cart for the checkout, you can add a `getTotal()` method, like this:

Listing 1-3

```
public function getTotal()
{
    $total = 0;
    foreach ($this->getItems() as $item)
    {
        $total += $item->getPrice() * $item->getQuantity();
    }

    return $total;
}
```

Using this method we are able to control the values returned from an object level. Imagine if later there is a decision to add some discount logic which affects the total - it can simply be added to the `getTotal()` method or even to the `getPrice()` methods of the items and the correct value would be returned.

Out of the box, symfony supports the two most popular open source ORMs in PHP: Propel and Doctrine. Symfony integrates both of them seamlessly. When creating a new symfony project, it's a matter of choice to use Propel or Doctrine.

This book will describe how to use the Propel and Doctrine objects, but for a more complete reference, a visit to the Propel¹³ website or the Doctrine¹⁴ website is recommended.

Rapid Application Development (RAD)

Programming web applications has long been a tedious and slow job. Following the usual software engineering life cycles (like the one proposed by the Rational Unified Process, for instance), the development of web applications could not start before a complete set of requirements was written, a lot of Unified Modeling Language (UML) diagrams were drawn, and tons of preliminary documentation was produced. This was due to the general speed of development, the lack of versatility of programming languages (you had to build, compile, restart, and who knows what else before actually seeing your program run), and most of all, to the fact that clients were quite reasonable and didn't change their minds constantly.

Today, business moves faster, and clients tend to constantly change their minds in the course of the project development. Of course, they expect the development team to adapt to their needs and modify the structure of an application quickly. Fortunately, the use of scripting languages like Python, Ruby, and PHP makes it easy to apply other programming strategies, such as rapid application development (RAD) or agile software development.

One of the ideas of these methodologies is to start developing as soon as possible so that the client can review a working prototype and offer additional direction. Then the application gets built in an iterative process, releasing increasingly feature-rich versions in short development cycles.

The consequences for the developer are numerous. A developer doesn't need to think about the future when implementing a feature. The method used should be as simple and straightforward as possible. This is well illustrated by the maxim of the KISS principle: Keep It Simple, Stupid.

When the requirements evolve or when a feature is added, existing code usually has to be partly rewritten. This process is called refactoring, and happens a lot in the course of a web application development. Code is moved to other places according to its nature. Duplicated

13. <http://www.propelorm.org/>

14. <http://www.doctrine-project.org/>

portions of code are refactored to a single place, thus applying the Don't Repeat Yourself (DRY) principle.

And to make sure that the application still runs when it changes constantly, it needs a full set of unit tests that can be automated. If well written, unit tests are a solid way to ensure that nothing is broken by adding or refactoring code. Some development methodologies even stipulate writing tests before coding—that's called test-driven development (TDD).



There are many other principles and good habits related to agile development. One of the most effective agile development methodologies is called Extreme Programming (abbreviated as XP), and the XP literature will teach you a lot about how to develop an application in a fast and effective way. A good starting place is the XP series books by Kent Beck (Addison-Wesley).

Symfony is the perfect tool for RAD. As a matter of fact, the framework was built by a web agency applying the RAD principle for its own projects. This means that learning to use symfony is not about learning a new language, but more about applying the right reflexes and the best judgment in order to build applications in a more effective way.

YAML

According to the official YAML website¹⁵, YAML is “a human friendly data serialization standard for all programming languages”. Put another way, YAML is a very simple language used to describe data in an XML-like way but with a much simpler syntax. It is especially useful to describe data that can be translated into arrays and hashes, like this:

```
$house = array(
    'family' => array(
        'name'      => 'Doe',
        'parents'   => array('John', 'Jane'),
        'children'  => array('Paul', 'Mark', 'Simone')
    ),
    'address' => array(
        'number'    => 34,
        'street'    => 'Main Street',
        'city'      => 'Nowheretown',
        'zipcode'   => '12345'
    )
);
```

Listing 1-4

This PHP array can be automatically created by parsing the YAML string:

```
house:
  family:
    name:      Doe
    parents:
      - John
      - Jane
    children:
      - Paul
      - Mark
      - Simone
  address:
    number: 34
    street: Main Street
```

Listing 1-5

15. <http://www.yaml.org/>

```
city: Nowheretown
zipcode: "12345"
```

In YAML, structure is shown through indentation, sequence items are denoted by a dash, and key/value pairs within a map are separated by a colon. YAML also has a shorthand syntax to describe the same structure with fewer lines, where arrays are explicitly shown with [] and hashes with {}. Therefore, the previous YAML data can be written in a shorter way, as follows:

Listing 1-6

```
house:
  family: { name: Doe, parents: [John, Jane], children: [Paul, Mark, Simone] }
  address: { number: 34, street: Main Street, city: Nowheretown, zipcode: "12345" }
```

YAML is an acronym for “YAML Ain’t Markup Language” and pronounced “yamel”. The format has been around since 2001, and YAML parsers exist for a large variety of languages.



The specifications of the YAML format are available at <http://www.yaml.org/>¹⁶.

As you can see, YAML is much faster to write than XML (no more closing tags or explicit quotes), and it is more powerful than .ini files (which don’t support hierarchy). That is why Symfony uses YAML as the preferred language to store configuration. You will see a lot of YAML files in this book, but it is so straightforward that you probably don’t need to learn more about it.

Summary

Symfony is a PHP web application framework. It adds a new layer on top of the PHP language, providing tools that speed up the development of complex web applications. This book will tell you all about it, and you just need to be familiar with the basic concepts of modern programming to understand it—namely object-oriented programming (OOP), object-relational mapping (ORM), and rapid application development (RAD). The only required technical background is knowledge of PHP.

16. <http://www.yaml.org/>

Chapter 2

Exploring Symfony's Code

At first glance, the code behind a symfony-driven application can seem quite daunting. It consists of many directories and scripts, and the files are a mix of PHP classes, HTML, and even an intermingling of the two. You'll also see references to classes that are otherwise nowhere to be found within the application folder, and the directory depth stretches to six levels. But once you understand the reason behind all of this seeming complexity, you'll suddenly feel like it's so natural that you wouldn't trade the symfony application structure for any other. This chapter explains away that intimidated feeling.

The MVC Pattern

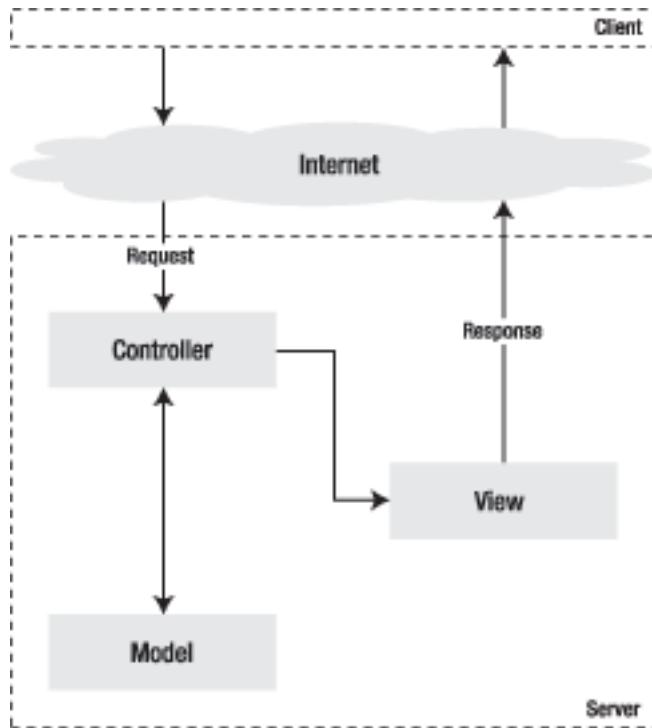
Symfony is based on the classic web design pattern known as the MVC architecture, which consists of three levels:

- 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.

Figure 2-1 illustrates the MVC pattern.

The MVC architecture separates the business logic (model) and the presentation (view), resulting in greater maintainability. For instance, if your application should run on both standard web browsers and handheld devices, you just need a new view; 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, and so on) 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.

Figure 2-1 - The MVC pattern



MVC Layering

To help you understand MVC's advantages, let's see how to convert a basic PHP application to an MVC-architected application. A list of posts for a weblog application will be a perfect example.

Flat Programming

In a flat PHP file, displaying a list of database entries might look like the script presented in Listing 2-1.

Listing 2-1 - A Flat Script

```

Listing 2-1 <?php

// Connecting, selecting database
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

// Performing SQL query
$result = mysql_query('SELECT date, title FROM post', $link);

?>

<html>
  <head>
    <title>List of Posts</title>
  </head>
  <body>
    <h1>List of Posts</h1>
    <table>
      <tr><th>Date</th><th>Title</th></tr>
<?php
// Printing results in HTML

```

```

while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
echo "\t<tr>\n";
printf("\t\t<td> %s </td>\n", $row['date']);
printf("\t\t<td> %s </td>\n", $row['title']);
echo "\t</tr>\n";
}
?>
    </table>
  </body>
</html>

<?php

// Closing connection
mysql_close($link);

?>

```

That's quick to write, fast to execute, and impossible to maintain. The following are the major problems with this code:

- There is no error-checking (what if the connection to the database fails?).
- HTML and PHP code are mixed, even interwoven together.
- The code is tied to a MySQL database.

Isolating the Presentation

The `echo` and `printf` calls in Listing 2-1 make the code difficult to read. Modifying the HTML code to enhance the presentation is a hassle with the current syntax. So the code can be split into two parts. First, the pure PHP code with all the business logic goes in a controller script, as shown in Listing 2-2.

Listing 2-2 - The Controller Part, in index.php

```

<?php
// Connecting, selecting database
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

// Performing SQL query
$result = mysql_query('SELECT date, title FROM post', $link);

// Filling up the array for the view
$post = array();
while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
    $post[] = $row;
}

// Closing connection
mysql_close($link);

// Requiring the view
require('view.php');

```

*Listing
2-2*

The HTML code, containing template-like PHP syntax, is stored in a view script, as shown in Listing 2-3.

Listing 2-3 - The View Part, in view.php

Listing 2-3

```
<html>
  <head>
    <title>List of Posts</title>
  </head>
  <body>
    <h1>List of Posts</h1>
    <table>
      <tr><th>Date</th><th>Title</th></tr>
      <?php foreach ($posts as $post): ?>
      <tr>
        <td><?php echo $post['date'] ?></td>
        <td><?php echo $post['title'] ?></td>
      </tr>
      <?php endforeach; ?>
    </table>
  </body>
</html>
```

A good rule of thumb to determine whether the view is clean enough is that it should contain only a minimum amount of PHP code, in order to be understood by an HTML designer without PHP knowledge. The most common statements in views are `echo`, `if/endif`, `foreach/endforeach`, and that's about all. Also, there should not be PHP code echoing HTML tags.

All the logic is moved to the controller script, and contains only pure PHP code, with no HTML inside. As a matter of fact, you should imagine that the same controller could be reused for a totally different presentation, perhaps in a PDF file or an XML structure.

Isolating the Data Manipulation

Most of the controller script code is dedicated to data manipulation. But what if you need the list of posts for another controller, say one that would output an RSS feed of the weblog posts? What if you want to keep all the database queries in one place, to avoid code duplication? What if you decide to change the data model so that the `post` table gets renamed `weblog_post`? What if you want to switch to PostgreSQL instead of MySQL? In order to make all that possible, you need to remove the data-manipulation code from the controller and put it in another script, called the model, as shown in Listing 2-4.

Listing 2-4 - The Model Part, in model.php

Listing 2-4

```
<?php

function getAllPosts()
{
    // Connecting, selecting database
    $link = mysql_connect('localhost', 'myuser', 'mypassword');
    mysql_select_db('blog_db', $link);

    // Performing SQL query
    $result = mysql_query('SELECT date, title FROM post', $link);

    // Filling up the array
    $posts = array();
    while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
```

```

{
    $posts[] = $row;
}

// Closing connection
mysql_close($link);

return $posts;
}

```

The revised controller is presented in Listing 2-5.

Listing 2-5 - The Controller Part, Revised, in index.php

```
<?php
// Requiring the model
require_once('model.php');

// Retrieving the list of posts
$posts = getAllPosts();

// Requiring the view
require('view.php');
```

*Listing
2-5*

The controller becomes easier to read. Its sole task is to get the data from the model and pass it to the view. In more complex applications, the controller also deals with the request, the user session, the authentication, and so on. The use of explicit names for the functions of the model even makes code comments unnecessary in the controller.

The model script is dedicated to data access and can be organized accordingly. All parameters that don't depend on the data layer (like request parameters) must be given by the controller and not accessed directly by the model. The model functions can be easily reused in another controller.

Layer Separation Beyond MVC

So the principle of the MVC architecture is to separate the code into three layers, according to its nature. Data logic code is placed within the model, presentation code within the view, and application logic within the controller.

Other additional design patterns can make the coding experience even easier. The model, view, and controller layers can be further subdivided.

Database Abstraction

The model layer can be split into a data access layer and a database abstraction layer. That way, data access functions will not use database-dependent query statements, but call some other functions that will do the queries themselves. If you change your database system later, only the database abstraction layer will need updating.

A sample database abstraction layer is presented in Listing 2-6, followed by an example of a MySQL-specific data access layer in Listing 2-7.

Listing 2-6 - The Database Abstraction Part of the Model

```
<?php
function open_connection($host, $user, $password)
{
```

*Listing
2-6*

```

    return mysql_connect($host, $user, $password);
}

function close_connection($link)
{
    mysql_close($link);
}

function query_database($query, $database, $link)
{
    mysql_select_db($database, $link);

    return mysql_query($query, $link);
}

function fetch_results($result)
{
    return mysql_fetch_array($result, MYSQL_ASSOC);
}

```

Listing 2-7 - The Data Access Part of the Model

Listing 2-7

```

function getAllPosts()
{
    // Connecting to database
    $link = open_connection('localhost', 'myuser', 'mypassword');

    // Performing SQL query
    $result = query_database('SELECT date, title FROM post', 'blog_db',
$link);

    // Filling up the array
    $posts = array();
    while ($row = fetch_results($result))
    {
        $posts[] = $row;
    }

    // Closing connection
    close_connection($link);

    return $posts;
}

```

You can check that no database-engine dependent functions can be found in the data access layer, making it database-independent. Additionally, the functions created in the database abstraction layer can be reused for many other model functions that need access to the database.



The examples in Listings 2-6 and 2-7 are still not very satisfactory, and there is some work left to do to have a full database abstraction (abstracting the SQL code through a database-independent query builder, moving all functions into a class, and so on). But the purpose of this book is not to show you how to write all that code by hand, and you will see in Chapter 8 that Symfony natively does all the abstraction very well.

View Elements

The view layer can also benefit from some code separation. A web page often contains consistent elements throughout an application: the page headers, the graphical layout, the footer, and the global navigation. Only the inner part of the page changes. That's why the view is separated into a layout and a template. The layout is usually global to the application, or to a group of pages. The template only puts in shape the variables made available by the controller. Some logic is needed to make these components work together, and this view logic layer will keep the name view. According to these principles, the view part of Listing 2-3 can be separated into three parts, as shown in Listings 2-8, 2-9, and 2-10.

Listing 2-8 - The Template Part of the View, in mytemplate.php

```
<h1>List of Posts</h1>
<table>
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
<tr>
    <td><?php echo $post['date'] ?></td>
    <td><?php echo $post['title'] ?></td>
</tr>
<?php endforeach; ?>
</table>
```

*Listing
2-8*

Listing 2-9 - The View Logic Part of the View

```
<?php
$title = 'List of Posts';
$posts = getAllPosts();
```

*Listing
2-9*

Listing 2-10 - The Layout Part of the View

```
<html>
  <head>
    <title><?php echo $title ?></title>
  </head>
  <body>
    <?php include('mytemplate.php'); ?>
  </body>
</html>
```

*Listing
2-10*

Action and Front Controller

The controller doesn't do much in the previous example, but in real web applications, the controller has a lot of work. An important part of this work is common to all the controllers of the application. The common tasks include request handling, security handling, loading the application configuration, and similar chores. This is why the controller is often divided into a front controller, which is unique for the whole application, and actions, which contain only the controller code specific to one page.

One of the great advantages of a front controller is that it offers a unique entry point to the whole application. If you ever decide to close the access to the application, you will just need to edit the front controller script. In an application without a front controller, each individual controller would need to be turned off.

Object Orientation

All the previous examples use procedural programming. The OOP capabilities of modern languages make the programming even easier, since objects can encapsulate logic, inherit from one another, and provide clean naming conventions.

Implementing an MVC architecture in a language that is not object-oriented raises namespace and code-duplication issues, and the overall code is difficult to read.

Object orientation allows developers to deal with such things as the view object, the controller object, and the model classes, and to transform all the functions in the previous examples into methods. It is a must for MVC architectures.



If you want to learn more about design patterns for web applications in an object-oriented context, read *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley, ISBN: 0-32112-742-0). Code examples in Fowler's book are in Java or C#, but are still quite readable for a PHP developer.

Symfony's MVC Implementation

Hold on a minute. For a single page listing the posts in a weblog, how many components are required? As illustrated in Figure 2-2, we have the following parts:

- Model layer
 - Database abstraction
 - Data access
- View layer
 - View
 - Template
 - Layout
- Controller layer
 - Front controller
 - Action

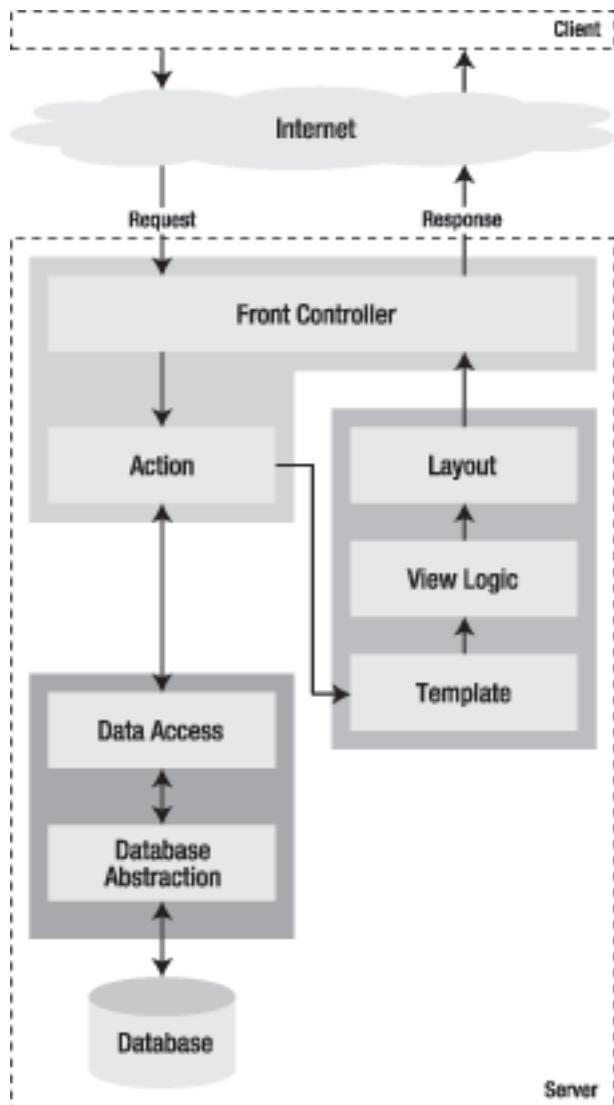
Seven scripts—a whole lot of files to open and to modify each time you create a new page! However, symfony makes things easy. While taking the best of the MVC architecture, symfony implements it in a way that makes application development fast and painless.

First of all, the front controller and the layout are common to all actions in an application. You can have multiple controllers and layouts, but you need only one of each. The front controller is pure MVC logic component, and you will never need to write a single one, because symfony will generate it for you.

The other good news is that the classes of the model layer are also generated automatically, based on your data structure. This is the job of the ORM library, which provides class skeletons and code generation. If the ORM finds foreign key constraints or date fields, it will provide special accessor and mutator methods that will make data manipulation a piece of cake. And the database abstraction is totally invisible to you, because it is handled natively by PHP Data Objects. So if you decide to change your database engine at anytime, you have zero code to refactor. You just need to change one configuration parameter.

And the last thing is that the view logic can be easily translated as a simple configuration file, with no programming needed.

Figure 2-2 - Symfony workflow



That means that the list of posts described in our example would require only three files to work in symfony, as shown in Listings 2-11, 2-12, and 2-13.

Listing 2-11 - list Action, in myproject/apps/myapp/modules/weblog/actions/actions.class.php

```
<?php
class weblogActions extends sfActions
{
    public function executeList()
    {
        $this->posts = PostPeer::doSelect(new Criteria());
    }
}
```

*Listing
2-11*

Listing 2-12 - list Template, in myproject/apps/myapp/modules/weblog/templates/listSuccess.php

```
<?php slot('title', 'List of Posts') ?>
<h1>List of Posts</h1>
<table>
```

*Listing
2-12*

```
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
<tr>
    <td><?php echo $post->getDate() ?></td>
    <td><?php echo $post->getTitle() ?></td>
</tr>
<?php endforeach; ?>
</table>
```

In addition, you will still need to define a layout, as shown in Listing 2-13, but it will be reused many times.

Listing 2-13 - Layout, in myproject/apps/myapp/templates/layout.php

Listing 2-13

```
<html>
    <head>
        <title><?php include_slot('title') ?></title>
    </head>
    <body>
        <?php echo $sf_content ?>
    </body>
</html>
```

And that is really all you need. This is the exact code required to display the very same page as the flat script shown earlier in Listing 2-1. The rest (making all the components work together) is handled by symfony. If you count the lines, you will see that creating the list of posts in an MVC architecture with symfony doesn't require more time or coding than writing a flat file. Nevertheless, it gives you huge advantages, notably clear code organization, reusability, flexibility, and much more fun. And as a bonus, you have XHTML conformance, debug capabilities, easy configuration, database abstraction, smart URL routing, multiple environments, and many more development tools.

Symfony Core Classes

The MVC implementation in symfony uses several classes that you will meet quite often in this book:

- `sfController` is the controller class. It decodes the request and hands it to the action.
- `sfRequest` stores all the request elements (parameters, cookies, headers, and so on).
- `sfResponse` contains the response headers and contents. This is the object that will eventually be converted to an HTML response and be sent to the user.
- The context (retrieved by `sfContext::getInstance()`) stores a reference to all the core objects and the current configuration; it is accessible from everywhere.

You will learn more about these objects in Chapter 6.

As you can see, all the symfony classes use the `sf` prefix, as do the symfony core variables in the templates. This should avoid name collisions with your own classes and variables, and make the core framework classes sociable and easy to recognize.



Among the coding standards used in symfony, UpperCamelCase is the standard for class and variable naming. Two exceptions exist: core symfony classes start with `sf`, which is lowercase, and variables found in templates use the underscore-separated syntax.

Code Organization

Now that you know the different components of a symfony application, you're probably wondering how they are organized. Symfony organizes code in a project structure and puts the project files into a standard tree structure.

Project Structure: Applications, Modules, and Actions

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 many mini-sites, with each site as a different application. Note that hyperlinks between applications must be in the absolute form.

Each application is a set of one or more modules. A module usually represents a page or a group of pages with a similar purpose. For example, you might have the modules `home`, `articles`, `help`, `shoppingCart`, `account`, and so on.

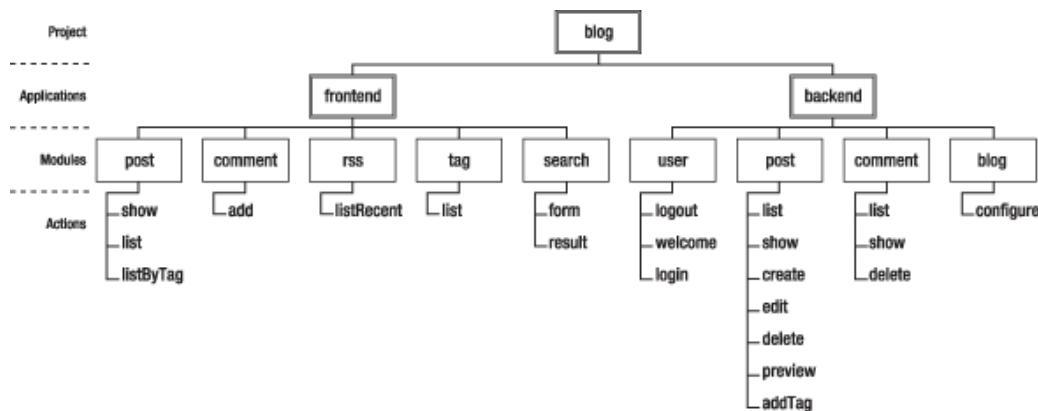
Modules hold actions, which represent the various actions that can be done in a module. For example, a `shoppingCart` module can have `add`, `show`, and `update` actions. Generally, actions can be described by a verb. Dealing with actions is almost like dealing with pages in a classic web application, although two actions can result in the same page (for instance, adding a comment to a post in a weblog will redisplay the post with the new comment).



If this represents too many levels for a beginning project, it is very easy to group all actions into one single module, so that the file structure can be kept simple. When the application gets more complex, it will be time to organize actions into separate modules. As mentioned in Chapter 1, rewriting code to improve its structure or readability (but preserving its behavior) is called refactoring, and you will do this a lot when applying RAD principles.

Figure 2-3 shows a sample code organization for a weblog project, in a project/application/module/action structure. But be aware that the actual file tree structure of the project will differ from the setup shown in the figure.

Figure 2-3 - Example of code organization



File Tree Structure

All web projects generally share the same types of contents, such as the following:

- A database, such as MySQL or PostgreSQL

- Static files (HTML, images, JavaScript files, style sheets, and so on)
- Files uploaded by the site users and administrators
- PHP classes and libraries
- Foreign libraries (third-party scripts)
- Batch files (scripts to be launched by a command line or via a cron table)
- Log files (traces written by the application and/or the server)
- Configuration files

Symfony provides a standard file tree structure to organize all these contents in a logical way, consistent with the architecture choices (MVC pattern and project/application/module grouping). This is the tree structure that is automatically created when initializing every project, application, or module. Of course, you can customize it completely, to reorganize the files and directories at your convenience or to match your client's requirements.

Root Tree Structure

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

Listing 2-14

```
apps/
  frontend/
  backend/
cache/
config/
data/
  sql/
doc/
lib/
  model/
log/
plugins/
test/
  bootstrap/
  unit/
  functional/
web/
  css/
  images/
  js/
  uploads/
```

Table 2-1 describes the contents of these directories.

Table 2-1 - Root Directories

Directory	Description
apps/	Contains one directory for each application of the project (typically, <code>frontend</code> and <code>backend</code> for the front and back office).
cache/	Contains the cached version of the configuration, and (if you activate it) the cache version of the actions and templates of the project. The cache mechanism (detailed in Chapter 12) uses these files to speed up the answer to web requests. Each application will have a subdirectory here, containing preprocessed PHP and HTML files.
config/	Holds the general configuration of the project.
data/	Here, you can store the data files of the project, like a database schema, a SQL file that creates tables, or even a SQLite database file.

Directory	Description
<code>doc/</code>	A default place to store project documentation
<code>lib/</code>	Dedicated to foreign classes or libraries. Here, you can add the code that needs to be shared among your applications. The <code>model/</code> subdirectory stores the object model of the project (described in Chapter 8).
<code>log/</code>	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. Symfony creates one log file per application and per environment (log files are discussed in Chapter 16).
<code>plugins/</code>	Stores the plug-ins installed in the application (plug-ins are discussed in Chapter 17).
<code>test/</code>	Contains unit and functional tests written in PHP and compatible with the symfony testing framework (discussed in Chapter 15). During the project setup, symfony automatically adds some stubs with a few basic tests.
<code>web/</code>	The root for the web server. The only files accessible from the Internet are the ones located in this directory.

Application Tree Structure

The tree structure of all application directories is the same:

```
apps/
 [application name]/
 config/
 i18n/
 lib/
 modules/
 templates/
 layout.php
```

*Listing
2-15*

Table 2-2 describes the application subdirectories.

Table 2-2 - Application Subdirectories

Directory	Description
<code>config/</code>	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'll learn more about application configuration in the Chapter 5.
<code>i18n/</code>	Contains files used for the internationalization of the application—mostly interface translation files (Chapter 13 deals with internationalization). You can bypass this directory if you choose to use a database for internationalization.
<code>lib/</code>	Contains classes and libraries that are specific to the application.
<code>modules/</code>	Stores all the modules that contain the features of the application.
<code>templates/</code>	Lists the global templates of the application—the ones that are shared by all modules. By default, it contains a <code>layout.php</code> file, which is the main layout in which the module templates are inserted.



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

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. You need to keep this last constraint in mind during initialization, when you choose how to divide your project into applications.

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:

Listing 2-16

```
apps/
[application name]/
  modules/
    [module name]/
      actions/
        actions.class.php
      config/
      lib/
      templates/
        indexSuccess.php
```

Table 2-3 describes the module subdirectories.

Table 2-3 - Module Subdirectories

Directory	Description
<code>actions/</code>	Generally contains a single class file named <code>actions.class.php</code> , in which you can store all the actions of the module. You can also write different actions of a module in separate files.
<code>config/</code>	Can contain custom configuration files with local parameters for the module.
<code>lib/</code>	Stores classes and libraries specific to the module.
<code>templates/</code>	Contains the templates corresponding to the actions of the module. A default template, called <code>indexSuccess.php</code> , is created during module setup.



The `config/`, and `lib/` directories are not created automatically for a new module so you will need to create them manually if required

Web Tree Structure

There are very few constraints for the `web` directory, which is the directory of publicly accessible files. Following a few basic naming conventions will provide default behaviors and useful shortcuts in the templates. Here is an example of a `web` directory structure:

Listing 2-17

```
web/
  css/
  images/
  js/
  uploads/
```

Conventionally, the static files are distributed in the directories listed in Table 2-4.

Table 2-4 - Typical Web Subdirectories

Directory Description

css/	Contains style sheets with a .css extension.
images/	Contains images with a .jpg, .png, or .gif extension.
js/	Contains JavaScript files with a .js extension.
uploads/	Can contain the files uploaded by the users. Even though the directory usually contains images, it is distinct from the images directory so that the synchronization of the development and production servers does not affect the uploaded images.



Even though it is highly recommended that you maintain the default tree structure, it is possible to modify it for specific needs, such as to allow a project to run on a server with different tree structure rules and coding conventions. Refer to Chapter 19 for more information about modifying the file tree structure.

Common Instruments

A few techniques are used repeatedly in symfony, and you will meet them quite often in this book and in your own projects. These include parameter holders, constants, and class autoloading.

Parameter Holders

Many of the symfony classes contain a parameter holder. It is a convenient way to encapsulate attributes with clean getter and setter methods. For instance, the `sfRequest` class holds a parameter holder that you can retrieve by calling the `getParameterHolder()` method. Each parameter holder stores data the same way, as illustrated in Listing 2-14.

Listing 2-14 - Using the sfRequest Parameter Holder

```
$request->getParameterHolder()->set('foo', 'bar');
echo $request->getParameterHolder()->get('foo');
=> 'bar'
```

Listing 2-18

Most of the classes using a parameter holder provide proxy methods to shorten the code needed for get/set operations. This is the case for the `sfRequest` object, so you can do the same as in Listing 2-14 with the code of Listing 2-15.

Listing 2-15 - Using the sfRequest Parameter Holder Proxy Methods

```
$request->setParameter('foo', 'bar');
echo $request->getParameter('foo');
=> 'bar'
```

Listing 2-19

The parameter holder getter accepts a default value as a second argument. This provides a useful fallback mechanism that is much more concise than possible with a conditional statement. See Listing 2-16 for an example.

Listing 2-16 - Using the Attribute Holder Getter's Default Value

Listing 2-20

```
// The 'foobar' parameter is not defined, so the getter returns an empty
value
echo $request->getParameter('foobar');
=> null

// A default value can be used by putting the getter in a condition
if ($request->hasParameter('foobar'))
{
    echo $request->getParameter('foobar');
}
else
{
    echo 'default';
}
=> default

// But it is much faster to use the second getter argument for that
echo $request->getParameter('foobar', 'default');
=> default
```

Some symfony core classes also use a parameter holder that supports namespaces (thanks to the `sfNamespacedParameterHolder` class). If you specify a third argument to a setter or a getter, it is used as a namespace, and the parameter will be defined only within that namespace. Listing 2-17 shows an example.

Listing 2-17 - Using the sfUser Parameter Holder Namespace

Listing 2-21

```
$user->setAttribute('foo', 'bar1');
$user->setAttribute('foo', 'bar2', 'my/name/space');
echo $user->getAttribute('foo');
=> 'bar1'
echo $user->getAttribute('foo', null, 'my/name/space');
=> 'bar2'
```

Of course, you can add a parameter holder to your own classes to take advantage of their syntax facilities. Listing 2-18 shows how to define a class with a parameter holder.

Listing 2-18 - Adding a Parameter Holder to a Class

Listing 2-22

```
class MyClass
{
    protected $parameterHolder = null;

    public function initialize($parameters = array())
    {
        $this->parameterHolder = new sfParameterHolder();
        $this->parameterHolder->add($parameters);
    }

    public function getParameterHolder()
    {
        return $this->parameterHolder;
    }
}
```

Constants

You will not find any constants in symfony because by their very nature you can't change their value once they are defined. Symfony uses its own configuration object, called `sfConfig`, which replaces constants. It provides static methods to access parameters from everywhere. Listing 2-19 demonstrates the use of `sfConfig` class methods.

Listing 2-19 - Using the sfConfig Class Methods Instead of Constants

```
// Instead of PHP constants,
define('FOO', 'bar');
echo FOO;

// symfony uses the sfConfig object
sfConfig::set('foo', 'bar');
echo sfConfig::get('foo');
```

*Listing
2-23*

The `sfConfig` methods support default values, and you can call the `sfConfig::set()` method more than once on the same parameter to change its value. Chapter 5 discusses `sfConfig` methods in more detail.

Class Autoloading

Usually, when you use a class method or create an object in PHP, you need to include the class definition first:

```
include_once 'classes/MyClass.php';
$myObject = new MyClass();
```

*Listing
2-24*

On large projects with many classes and a deep directory structure, keeping track of all the class files to include and their paths can be time consuming. By providing an `spl_autoload_register()` function, symfony makes `include_once` statements unnecessary, and you can write directly:

```
$myObject = new MyClass();
```

*Listing
2-25*

Symfony will then look for a `MyClass` definition in all files ending with `class.php` in one of the project's `lib/` directories. If the class definition is found, it will be included automatically.

So if you store all your classes in `lib/` directories, you don't need to include classes anymore. That's why symfony projects usually do not contain any `include_once` or `require_once` statements.



For better performance, the symfony autoloader scans a list of directories (defined in an internal configuration file) during the first request. It then registers all the classes these directories contain and stores the class/file correspondence in a PHP file as an associative array. That way, future requests don't need to do the directory scan anymore. This is why you need to clear the cache every time you add or move a class file in your project by calling the `symfony cache:clear` command (except in the development environment, where symfony clears the cache when it cannot find a class). You will learn more about the cache in Chapter 12, and about the autoloading configuration in Chapter 19.

Summary

Using an MVC framework forces you to divide and organize your code according to the framework's conventions. Presentation code goes to the view, data manipulation code goes to the model, and the request manipulation logic goes to the controller. This makes applying the MVC pattern both very helpful and at the same time quite restrictive.

Symfony is an MVC framework written in PHP. Its structure is designed to get the best out of the MVC pattern, whilst maintaining great ease of use. Thanks to its versatility and configurability, symfony is suitable for all web application projects.

Now that you understand the underlying theory behind symfony, you are almost ready to develop your first application. But before that, you need a symfony installation up and running on your development server.

Chapter 3

Running Symfony

As you've learned in previous chapters, the symfony framework is a set of files written in PHP. A symfony project uses these files, so installing symfony means getting these files and making them available for the project.

Symfony requires at least PHP 5.2.4. Make sure you have it installed by opening a command line and typing this command:

```
$ php -v
```

Listing 3-1

```
PHP 5.3.1 (cli) (built: Jan 6 2010 20:54:10)
Copyright (c) 1997-2009 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2009 Zend Technologies
```

If the version number is 5.2.4 or higher, then you're ready for the installation, as described in this chapter.

Prerequisites

Before installing symfony, you need to check that your computer has everything installed and configured correctly. Take the time to conscientiously read this chapter and follow all the steps required to check your configuration, as it may save your day further down the road.

Third-Party Software

First of all, you need to check that your computer has a friendly working environment for web development. At a minimum, you need a web server (Apache, for instance), a database engine (MySQL, PostgreSQL, SQLite, or any PDO¹⁷-compatible database engine), and PHP 5.2.4 or later.

Command Line Interface

The symfony framework comes bundled with a command line tool that automates a lot of work for you. If you are a Unix-like OS user, you will feel right at home. If you run a Windows system, it will also work fine, but you will just have to type a few commands at the cmd prompt.

17. <http://www.php.net/PDO>



Unix shell commands can come in handy in a Windows environment. If you would like to use tools like `tar`, `gzip` or `grep` on Windows, you can install Cygwin¹⁸. The adventurous may also like to try Microsoft's Windows Services for Unix¹⁹.

PHP Configuration

As PHP configurations can vary a lot from one OS to another, or even between different Linux distributions, you need to check that your PHP configuration meets the symfony minimum requirements.

First, ensure that you have PHP 5.2.4 as a minimum installed by using the `phpinfo()` built-in function or by running `php -v` on the command line. Be aware that on some configurations, you might have two different PHP versions installed: one for the command line, and another for the web.

Then, download the symfony configuration checker script at the following URL:

Listing 3-2 <http://sf-to.org/1.4/check.php>

Save the script somewhere under your current web root directory.

Launch the configuration checker script from the command line:

Listing 3-3 `$ php check_configuration.php`

If there is a problem with your PHP configuration, the output of the command will give you hints on what to fix and how to fix it.

You should also execute the checker from a browser and fix the issues it might discover. That's because PHP can have a distinct `php.ini` configuration file for these two environments, with different settings.



Don't forget to remove the file from your web root directory afterwards.



If your goal is to give symfony a try for a few hours, you can install the symfony sandbox as described at the end of this chapter. If you want to bootstrap a real world project or want to learn more about symfony, keep reading.

Symfony Installation

Initializing the Project Directory

Before installing symfony, you first need to create a directory that will host all the files related to your project:

Listing 3-4 `$ mkdir -p /home/sfproject
$ cd /home/sfproject`

Or on Windows:

18. <http://cygwin.com/>

19. <http://technet.microsoft.com/en-gb/interopmigration/bb380242.aspx>

```
c:\> mkdir c:\dev\sfproject
c:\> cd c:\dev\sfproject
```

*Listing
3-5*



Windows users are advised to run symfony and to setup their new project in a path which contains no spaces. Avoid using the Documents and Settings directory, including anywhere under My Documents.



If you create the symfony project directory under the web root directory, you won't need to configure your web server. Of course, for production environments, we strongly advise you to configure your web server as explained in the web server configuration section.

Choosing the Symfony Version

Now, you need to install symfony. As the symfony framework has several stable versions, you need to choose the one you want to install by reading the installation page²⁰ on the symfony website.

Choosing the Symfony Installation Location

You can install symfony globally on your machine, or embed it into each of your projects. The latter is the recommended approach as projects will then be totally independent from each other and upgrading your locally installed symfony won't break some of your projects unexpectedly. It also means you will be able to have projects using different versions of symfony, and upgrade them one at a time as you see fit.

As a best practice, many people install the symfony framework files in the lib/vendor project directory. So, first, create this directory:

```
$ mkdir -p lib/vendor
```

*Listing
3-6*

Installing Symfony

Installing from an archive

The easiest way to install symfony is to download the archive for the version you choose from the symfony website. Go to the installation page for the version you have just chosen, symfony 1.4²¹ for instance.

Under the “**Source Download**” section, you will find the archive in .tgz or in .zip format. Download the archive, put it under the freshly created lib/vendor/ directory, un-archive it, and rename the directory to symfony:

```
$ cd lib/vendor
$ tar zxpf symfony-1.4.0.tgz
$ mv symfony-1.4.0 symfony
$ rm symfony-1.4.0.tgz
```

*Listing
3-7*

Under Windows, unzipping the zip file can be achieved using Windows Explorer. After you rename the directory to symfony, there should be a directory structure similar to c:\dev\sfproject\lib\vendor\symfony.

20. <http://www.symfony-project.org/installation>

21. http://www.symfony-project.org/installation/1_4

Installing from Subversion (recommended)

If your project use Subversion, it is even better to use the `svn:externals` property to embed symfony into your project in the `lib/vendor/` directory:

Listing 3-8 `$ svn pe svn:externals lib/vendor/`

If everything goes well, this command will run your favorite editor to give you the opportunity to configure the external Subversion sources.



On Windows, you can use tools like TortoiseSVN²² to do everything without the need to use the console.

If you are conservative, tie your project to a specific release (a subversion tag):

Listing 3-9 `symfony http://svn.symfony-project.com/tags/RELEASE_1_4_0`

Whenever a new release comes out (as announced on the symfony blog²³), you will need to change the URL to the new version.

If you want to go the bleeding-edge route, use the 1.4 branch:

Listing 3-10 `symfony http://svn.symfony-project.com/branches/1.4/`

Using the branch makes your project benefit from bug fixes automatically whenever you run an `svn update`.

Installation Verification

Now that symfony is installed, check that everything is working by using the `symfony` command line to display the symfony version (note the capital V):

Listing 3-11 `$ cd ../../`
`$ php lib/vendor/symfony/data/bin/symfony -V`

On Windows:

Listing 3-12 `c:\> cd ..\..`
`c:\> php lib\vendor\symfony\data\bin\symfony -V`

After you have created your project (below), running this command also displays the path to the symfony installation directory, which is stored in `config/ProjectConfiguration.class.php`.

If when you check this, the path to symfony is an absolute one (which should not be by default if you follow the below instructions), change it so it reads like follows for better portability:

Listing 3-13 `// config/ProjectConfiguration.class.php`
`require_once dirname(__FILE__).'../../lib/vendor/symfony/lib/autoload/sfCoreAutoload.class.php';`

That way, you can move the project directory anywhere on your machine or another one, and it will just work.

22. <http://tortoisessvn.net/>

23. <http://www.symfony-project.org/blog/>



If you are curious about what this command line tool can do for you, type `symfony` to list the available options and tasks:

```
$ php lib/vendor/symfony/data/bin/symfony
```

*Listing
3-14*

On Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony
```

*Listing
3-15*

The `symfony` command line is the developer's best friend. It provides a lot of utilities that improve your productivity for day-to-day activities like cleaning the cache, generating code, and much more.

Project Setup

In `symfony`, **applications** sharing the same data model are regrouped into **projects**. For most projects, you will have two different applications: a frontend and a backend.

Project Creation

From the `sfproject/` directory, run the `symfony generate:project` task to actually create the `symfony` project:

```
$ php lib/vendor/symfony/data/bin/symfony generate:project PROJECT_NAME
```

*Listing
3-16*

On Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony generate:project PROJECT_NAME
```

*Listing
3-17*

The `generate:project` task generates the default structure of directories and files needed for a `symfony` project.



Why does `symfony` generate so many files? One of the main benefits of using a full-stack framework is to standardize your development. Thanks to `symfony`'s default structure of files and directories, any developer with some `symfony` knowledge can take over the maintenance of any `symfony` project. In a matter of minutes, he will be able to dive into the code, fix bugs, and add new features.

The `generate:project` task has also created a `symfony` shortcut in the project root directory to shorten the number of characters you have to write when running a task.

So, from now on, instead of using the fully qualified path to the `symfony` program, you can use the `symfony` shortcut.

Configuring the Database

The `symfony` framework supports all PDO²⁴-supported databases (MySQL, PostgreSQL, SQLite, Oracle, MSSQL, ...) out of the box. On top of PDO, `symfony` comes bundled with two ORM tools: Propel and Doctrine.

When creating a new project, Doctrine is enabled by default. Configuring the database used by Doctrine is as simple as using the `configure:database` task:

24. <http://www.php.net/PDO>

Listing 3-18

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

The `configure:database` task takes three arguments: the PDO DSN²⁵, the username, and the password to access the database. If you don't need a password to access your database on the development server, just omit the third argument.



If you want to use Propel instead of Doctrine, add `--orm=Propel` when creating the project with the `generate:project` task. And if you don't want to use an ORM, just pass `--orm=none`.

Application Creation

Now, create the frontend application by running the `generate:app` task:

Listing 3-19

```
$ php symfony generate:app frontend
```



Because the `symfony` shortcut file is executable, Unix users can replace all occurrences of `'php symfony'` by `'./symfony'` from now on.

On Windows you can copy the `'symfony.bat'` file to your project and use `'symfony'` instead of `'php symfony'`:

Listing 3-20

```
c:\> copy lib\vendor\symfony\data\bin\symfony.bat .
```

Based on the application name given as an *argument*, the `generate:app` task creates the default directory structure needed for the application under the `apps/frontend/` directory.

Security

By default, the `generate:app` task has secured our application from the two most widespread vulnerabilities found on the web. That's right, `symfony` automatically takes security measures on our behalf.

To prevent XSS attacks, output escaping has been enabled; and to prevent CSRF attacks, a random CSRF secret has been generated.

Of course, you can tweak these settings thanks to the following *options*:

- `--escaping-strategy`: Enables or disables output escaping
- `--csrf-secret`: Enables session tokens in forms

If you know nothing about XSS²⁶ or CSRF²⁷, take the time to learn more about these security vulnerabilities.

Directory Structure Rights

Before trying to access your newly created project, you need to set the write permissions on the `cache/` and `log/` directories to the appropriate levels, so that both your web server and command line user can write to them:

25. <http://www.php.net/manual/en/pdo.drivers.php>

26. http://en.wikipedia.org/wiki/Cross-site_scripting

27. <http://en.wikipedia.org/wiki/CSRF>

```
$ symfony project:permissions
```

*Listing
3-21*

Tips for People using an SCM Tool

symfony only ever writes to two directories of a symfony project, `cache/` and `log/`. The content of these directories should be ignored by your SCM (by editing the `svn:ignore` property if you use Subversion for instance).

Web Server Configuration

The ugly Way

In the previous chapters, you have created a directory that hosts the project. If you have created it somewhere under the web root directory of your web server, you can already access the project in a web browser.

Of course, as there is no configuration, it is very fast to set up, but try to access the `config/` `databases.yml` file in your browser to understand the bad consequences of such a lazy attitude. If the user knows that your website is developed with symfony, he will have access to a lot of sensitive files.

Never ever use this setup on a production server, and read the next section to learn how to configure your web server properly.

The secure Way

A good web practice is to only put the files that need to be accessed by a web browser under the web root directory , such as stylesheets, JavaScripts and images. By default, we recommend storing these files under the `web/` sub-directory of a symfony project.

If you have a look at this directory, you will find some sub-directories for web assets (`css/` and `images/`) and the two front controller files. The front controllers are the only PHP files that need to be under the web root directory. All other PHP files can be hidden from the browser, which is a good idea as far as security is concerned.

Web Server Configuration

Now it is time to change your Apache configuration, to make the new project accessible to the world.

Locate and open the `httpd.conf` configuration file and add the following configuration at the end:

```
# Be sure to only have this line once in your configuration
NameVirtualHost 127.0.0.1:8080

# This is the configuration for your project
Listen 127.0.0.1:8080

<VirtualHost 127.0.0.1:8080>
    DocumentRoot "/home/sfproject/web"
    DirectoryIndex index.php
    <Directory "/home/sfproject/web">
        AllowOverride All
        Allow from All
```

*Listing
3-22*

```
</Directory>

Alias /sf /home/sfproject/lib/vendor/symfony/data/web/sf
<Directory "/home/sfproject/lib/vendor/symfony/data/web/sf">
    AllowOverride All
    Allow from All
</Directory>
</VirtualHost>
```



The `/sf` alias gives you access to images and javascript files needed to properly display default symfony pages and the web debug toolbar.

On Windows, you need to replace the `Alias` line with something like:

Listing 3-23 Alias /sf "c:\dev\sfproject\lib\vendor\symfony\data\web\sf"

And `/home/sfproject/web` should be replaced with:

Listing 3-24 c:\dev\sfproject\web

This configuration makes Apache listen to port `8080` on your machine, so the website will be accessible at the following URL:

Listing 3-25 `http://localhost:8080/`

You can change `8080` to any number, but favour numbers greater than `1024` as they do not require administrator rights.

Configure a dedicated Domain Name

If you are an administrator on your machine, it is better to setup virtual hosts instead of adding a new port each time you start a new project. Instead of choosing a port and adding a `Listen` statement, choose a domain name (for instance the real domain name with `.localhost` added at the end) and add a `ServerName` statement:

Listing 3-26 # This is the configuration for your project
<VirtualHost 127.0.0.1:80>
 ServerName www.myproject.com.localhost
 <!-- same configuration as before -->
</VirtualHost>

The domain name `www.myproject.com.localhost` used in the Apache configuration has to be declared locally. If you run a Linux system, it has to be done in the `/etc/hosts` file. If you run Windows XP, Vista or Win7, this file is located in the `C:\WINDOWS\system32\drivers\etc\` directory.

Add the following line:

Listing 3-27 127.0.0.1 www.myproject.com.localhost

Test the New Configuration

Restart Apache, and check that you now have access to the new application by opening a browser and typing `http://localhost:8080/index.php/`, or

`http://www.myproject.com.localhost/index.php/` depending on the Apache configuration you chose in the previous section.



If you have the Apache `mod_rewrite` module installed, you can remove the `index.php/` part of the URL. This is possible thanks to the rewriting rules configured in the `web/.htaccess` file.

You should also try to access the application in the development environment (see the next section for more information about environments). Type in the following URL:

`http://www.myproject.com.localhost/frontend_dev.php/`

Listing 3-28

The web debug toolbar should show in the top right corner, including small icons proving that your `sf/` alias configuration is correct.



The setup is a little different if you want to run symfony on an IIS server in a Windows environment. Find how to configure it in the related tutorial²⁸.

Using the Sandbox

If your goal is to give symfony a try for a few hours, keep reading this chapter as we will show you the fastest way to get you started.

The fastest way to experiment with symfony is to install the symfony sandbox. The sandbox is a dead-easy-to-install pre-packaged symfony project, already configured with some sensible defaults. It is a great way to practice using symfony without the hassle of a proper installation that respects the web best practices.



As the sandbox is pre-configured to use SQLite as a database engine, you need to check that your PHP supports SQLite. You can also read the Configuring the Database section to learn how to change the database used by the sandbox.

You can download the symfony sandbox in `.tgz` or `.zip` format from the symfony installation page²⁹ or at the following URLs:

Listing 3-29 http://www.symfony-project.org/get/sf_sandbox_1_4.tgz

http://www.symfony-project.org/get/sf_sandbox_1_4.zip

Un-archive the files somewhere under your web root directory, and you are done. Your symfony project is now accessible by requesting the `web/index.php` script from a browser.

28. http://www.symfony-project.org/more-with-symfony/1_4/en/11-Windows-and-Symfony

29. http://www.symfony-project.org/installation/1_4



Having all the symfony files under the web root directory is fine for testing symfony on your local computer, but is a really bad idea for a production server as it potentially makes all the internals of your application visible to end users.

You can now finish your installation by reading the Web Server Configuration section.



As a sandbox is just a normal symfony project where some tasks have been executed for you and some configuration changed, it is quite easy to use it as a starting point for a new project. However, keep in mind that you will probably need to adapt the configuration; for instance changing the security related settings (see the configuration of XSS and CSRF in this chapter).

Summary

To test and play with symfony on your local server, your best option for installation is definitely the sandbox, which contains a preconfigured symfony environment.

For a real development or on a production server, opt for the archive installation or the SVN checkout. This will install the symfony libraries, and you still need to initialize a project and an application. The last step of the application setup is the server configuration, which can be done in many ways. Symfony works perfectly fine with a virtual host, and it is the recommended solution.

If you have any problems during installation, you will find many tutorials and answers to frequently asked questions on the symfony website. If necessary, you can submit your problem to the symfony community, and you will get a quick and effective answer.

Once your project is initialized, it is a good habit to start a version-control process.

Now that you are ready to use symfony, it is time to see how to build a basic web application.

Chapter 4

The Basics Of Page Creation

Curiously, the first tutorial that programmers follow when learning a new language or a framework is the one that displays “Hello, world!” on the screen. It is strange to think of the computer as something that can greet the whole world, since every attempt in the artificial intelligence field has so far resulted in poor conversational abilities. But symfony isn’t dumber than any other program, and the proof is, you can create a page that says “Hello, <Your Name Here>” with it.

This chapter will teach you how to create a module, which is a structural element that groups pages. You will also learn how to create a page, which is divided into an action and a template, because of the MVC pattern. Links and forms are the basic web interactions; you will see how to insert them in a template and handle them in an action.

Creating a Module Skeleton

As Chapter 2 explained, symfony groups pages into modules. Before creating a page, you need to create a module, which is initially an empty shell with a file structure that symfony can recognize.

The symfony command line automates the creation of modules. You just need to call the `generate:module` task with the application name and the module name as arguments. In the previous chapter, you created a `frontend` application. To add a `content` module to this application, type the following commands:

Listing 4-1

```
$ cd ~/myproject
$ php symfony generate:module frontend content

>> dir+      ~/myproject/apps/frontend/modules/content/actions
>> file+     ~/myproject/apps/frontend/modules/content/actions/
actions.class.php
>> dir+      ~/myproject/apps/frontend/modules/content/templates
>> file+     ~/myproject/apps/frontend/modules/content/templates/
indexSuccess.php
>> file+     ~/myproject/test/functional/frontend/contentActionsTest.php
>> tokens    ~/myproject/test/functional/frontend/contentActionsTest.php
>> tokens    ~/myproject/apps/frontend/modules/content/actions/
actions.class.php
>> tokens    ~/myproject/apps/frontend/modules/content/templates/
indexSuccess.php
```

Apart from the `actions/`, and `templates/` directories, this command created only three files. The one in the `test/` folder concerns functional tests, and you don’t need to bother with

it until Chapter 15. The `actions.class.php` (shown in Listing 4-1) forwards to the default module congratulation page. The `templates/indexSuccess.php` file is empty.

Listing 4-1 - The Default Generated Action, in actions/actions.class.php

```
<?php  
  
class contentActions extends sfActions  
{  
    public function executeIndex()  
    {  
        $this->forward('default', 'module');  
    }  
}
```

Listing 4-2

 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 using PHP comments to document your project and prepares each class file to be compatible with the `phpDocumentor` tool³⁰.

For each new module, symfony creates a default `index` action. It is composed of an action method called `executeIndex` and a template file called `indexSuccess.php`. The meanings of the `execute` prefix and `Success` suffix will be explained in Chapters 6 and 7, respectively. In the meantime, you can consider that this naming is a convention. You can see the corresponding page (reproduced in Figure 4-1) by browsing to the following URL:

`http://localhost/frontend_dev.php/content/index`

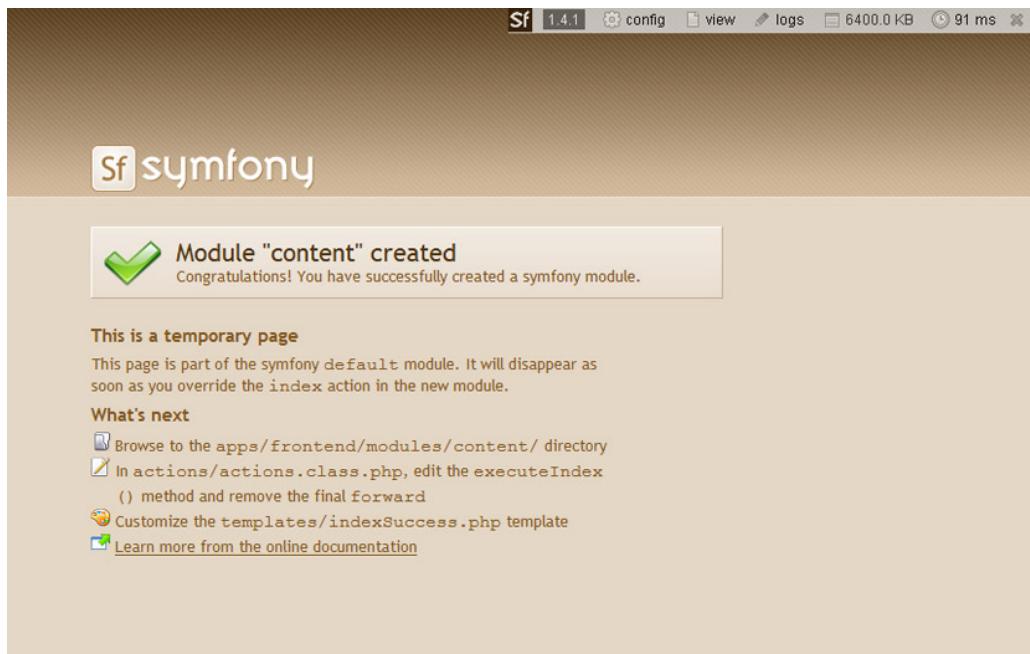
Listing 4-3

The default `index` action will not be used in this chapter, so you can remove the `executeIndex()` method from the `actions.class.php` file, and delete the `indexSuccess.php` file from the `templates/` directory.

 Symfony offers other ways to initiate a module than the command line. One of them is to create the directories and files yourself. In many cases, actions and templates of a module are meant to manipulate data of a given table. As the necessary code to create, retrieve, update, and delete records from a table is often the same, symfony provides a mechanism to generate this code for you.

Figure 4-1 - The default generated index page

30. <http://www.phpdoc.org/>



Adding a Page

In symfony, the logic behind pages is stored in the action, and the presentation is in templates. Pages without logic (still) require an empty action.

Adding an Action

The “Hello, world!” page will be accessible through a `show` action. To create it, just add an `executeShow` method to the `contentActions` class, as shown in Listing 4-2.

Listing 4-2 - Adding an Action Is Like Adding an Execute Method to the Action Class

```
Listing 4-4 <?php
class contentActions extends sfActions
{
    public function executeShow()
    {
    }
}
```

The name of the action method is always `executeXxx()`, where the second part of the name is the action name with the first letter capitalized.

Now, if you request the following URL:

```
Listing 4-5 http://localhost/frontend_dev.php/content/show
```

symfony will complain that the `showSuccess.php` template is missing. That’s normal; in symfony, a page is always made of an action and a template.



URLs (not domain names) are case-sensitive, and so is symfony (even though the method names are case-insensitive in PHP). This means that symfony will return a 404 error if you call `sHow` with the browser.

URLs are part of the response

Symfony contains a routing system that allows you to have a complete separation between the actual action name and the form of the URL needed to call it. This allows for custom formatting of the URL as if it were part of the response. You are no longer limited by the file structure nor by the request parameters; the URL for an action can look like the phrase you want. For instance, the call to the index action of a module called article usually looks like this:

```
http://localhost/frontend_dev.php/article/index?id=123
```

*Listing
4-6*

This URL retrieves a given article from a database. In this example, it retrieves an article (with `id=123`) in the Europe section that specifically discusses finance in France. But the URL can be written in a completely different way with a simple change in the `routing.yml` configuration file:

```
http://localhost/articles/europe/france/finance.html
```

*Listing
4-7*

Not only is the resulting URL search engine-friendly, it is also significant for the user, who can then use the address bar as a pseudo command line to do custom queries, as in the following:

```
http://localhost/articles/tagged/finance+france+euro
```

*Listing
4-8*

Symfony knows how to parse and generate smart URLs for the user. The routing system automatically peels the request parameters from a smart URL and makes them available to the action. It also formats the hyperlinks included in the response so that they look “smart”. You will learn more about this feature in Chapter 9.

Overall, this means that the way you name the actions of your applications should not be influenced by the way the URL used to call them should look, but by the actions’ functions in the application. An action name explains what the action actually does, and it is often a verb in the infinitive form (like `show`, `list`, `edit`, and so on). Action names can be made totally invisible to the end user, so don’t hesitate to use explicit action names (like `listByName` or `showWithComments`). You will economize on code comments to explain your action function, plus the code will be much easier to read.

Adding a Template

The action expects a template to render itself. A template is a file located in the `templates/` directory of a module, named by the action and the action termination. The default action termination is a “success,” so the template file to be created for the `show` action is to be called `showSuccess.php`.

Templates are supposed to contain only presentational code, so keep as little PHP code in them as possible. As a matter of fact, a page displaying “Hello, world!” can have a template as simple as the one in Listing 4-3.

Listing 4-3 - The content/templates/showSuccess.php Template

```
<p>Hello, world!</p>
```

*Listing
4-9*

If you need to execute some PHP code in the template, you should avoid using the usual PHP syntax, as shown in Listing 4-4. Instead, write your templates using the PHP alternative syntax, as shown in Listing 4-5, to keep the code understandable for non-PHP programmers. Not only will the final code be correctly indented, but it will also help you keep the complex PHP code in the action, because only control statements (`if`, `foreach`, `while`, and so on) have an alternative syntax.

Listing 4-4 - The Usual PHP Syntax, Good for Actions, But Bad for Templates

Listing 4-10

```
<p>Hello, world!</p>
<?php

if ($test)
{
    echo "<p>".time()."</p>";
}

?>
```

Listing 4-5 - The Alternative PHP Syntax, Good for Templates

Listing 4-11

```
<p>Hello, world!</p>
<?php if ($test): ?>
    <p><?php echo time(); ?></p>
<?php endif; ?>
```



A good rule of thumb to check if the template syntax is readable enough is that the file should not contain HTML code echoed by PHP or curly brackets. And most of the time, when opening a `<?php`, you will close it with `?>` in the same line.

Passing Information from the Action to the Template

The job of the action is to do all the complicated calculation, data retrieval, and tests, and to set variables for the template to be echoed or tested. Symfony makes the attributes of the action class (accessed via `$this->variableName` in the action) directly accessible to the template in the global namespace (via `$variableName`). Listings 4-6 and 4-7 show how to pass information from the action to the template.

Listing 4-6 - Setting an Action Attribute in the Action to Make It Available to the Template

Listing 4-12

```
<?php

class contentActions extends sfActions
{
    public function executeShow()
    {
        $today = getdate();
        $this->hour = $today['hours'];
    }
}
```

Listing 4-7 - The Template Has Direct Access to the Action Attributes

Listing 4-13

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
    <p>Or should I say good evening? It is already <?php echo $hour ?>. </p>
<?php endif; ?>
```

Note that the use of the short opening tags (`<?=`, equivalent to `<?php echo`) is not recommended for professional web applications, since your production web server may be able to understand more than one scripting language and consequently get confused. Besides, the short opening tags do not work with the default PHP configuration and need

server tweaking to be activated. Ultimately, when you have to deal with XML and validation, it falls short because `<?` has a special meaning in XML.



The template already has access to a few pieces of data without the need of any variable setup in the action. Every template can call methods of the `$sf_request`, `$sf_params`, `$sf_response`, and `$sf_user` objects. They contain data related to the current request, request parameters, response, and session. You will soon learn how to use them efficiently.

Linking to Another Action

You already know that there is a total decoupling between an action name and the URL used to call it. So if you create a link to `update` in a template as in Listing 4-8, it will only work with the default routing. If you later decide to change the way the URLs look, then you will need to review all templates to change the hyperlinks.

Listing 4-8 - Hyperlinks, the Classic Way

```
<a href="/frontend_dev.php/content/update?name=anonymous">
    I never say my name
</a>
```

*Listing
4-14*

To avoid this hassle, you should always use the `link_to()` helper to create hyperlinks to your application's actions. And if you only want to generate the URL part, the `url_for()` is the helper you're looking for.

A helper is a PHP function defined by symfony that is meant to be used within templates. It outputs some HTML code and is faster to use than writing the actual HTML code by yourself. Listing 4-9 demonstrates the use of the hyperlink helpers.

Listing 4-9 - The link_to(), and url_for() Helpers

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
    <p>Or should I say good evening? It is already <?php echo $hour ?>. </p>
<?php endif; ?>
<form method="post" action="<?php echo url_for('content/update') ?>">
    <label for="name">What is your name?</label>
    <input type="text" name="name" id="name" value="" />
    <input type="submit" value="Ok" />
    <?php echo link_to('I never say my name', 'content/
update?name=anonymous') ?>
</form>
```

*Listing
4-15*

The resulting HTML will be the same as previously, except that when you change your routing rules, all the templates will behave correctly and reformat the URLs accordingly.

Form manipulation deserves a whole chapter of its own, since symfony provides many tools to make it even easier. You will learn more about these helpers in Chapter 10.

The `link_to()` helper, like many other helpers, accepts another argument for special options and additional tag attributes. Listing 4-10 shows an example of an option argument and the resulting HTML. The option argument is either an associative array or a simple string showing `key=value` couples separated by blanks.

Listing 4-10 - Most Helpers Accept an Option Argument

```
// Option argument as an associative array
<?php echo link_to('I never say my name', 'content/update?name=anonymous',
```

*Listing
4-16*

```

array(
    'class'      => 'special_link',
    'confirm'    => 'Are you sure?',
    'absolute'   => true
)) ?>

// Option argument as a string
<?php echo link_to('I never say my name', 'content/update?name=anonymous',
    'class=special_link confirm=Are you sure? absolute=true') ?>

// Both calls output the same
=> <a class="special_link" onclick="return confirm('Are you sure?');"
    href="http://localhost/frontend_dev.php/content/update/name/anonymous">
    I never say my name</a>

```

Whenever you use a symfony helper that outputs an HTML tag, you can insert additional tag attributes (like the `class` attribute in the example in Listing 4-10) in the option argument. You can even write these attributes in the “quick-and-dirty” HTML 4.0 way (without double quotes), and symfony will output them in nicely formatted XHTML. That’s another reason why helpers are faster to write than HTML.



Because it requires an additional parsing and transformation, the string syntax is a little slower than the array syntax.

Like all symfony helpers, the link helpers are numerous and have many options. Chapter 9 will describe them in detail.

Getting Information from the Request

Whether the user sends information via a form (usually in a POST request) or via the URL (GET request), you can retrieve the related data from the action with the `getParameter()` method of the `sfRequest` object. Listing 4-11 shows how, in `update`, you retrieve the value of the `name` parameter.

Listing 4-11 - Getting Data from the Request Parameter in the Action

Listing 4-11

```

<?php

class contentActions extends sfActions
{
    // ...

    public function executeUpdate($request)
    {
        $this->name = $request->getParameter('name');
    }
}

```

As a convenience, all `executeXxx()` methods take the current `sfRequest` object as its first argument.

If the data manipulation is simple, you don’t even need to use the action to retrieve the request parameters. The template has access to an object called `$sf_params`, which offers a `get()` method to retrieve the request parameters, just like the `getParameter()` in the action.

If `executeUpdate()` were empty, Listing 4-12 shows how the `updateSuccess.php` template would retrieve the same `name` parameter.

Listing 4-12 - Getting Data from the Request Parameter Directly in the Template

```
<p>Hello, <?php echo $sf_params->get('name') ?>!</p>
```

*Listing
4-18*



Why not use the `$_POST`, `$_GET`, or `$_REQUEST` variables instead? Because then your URLs will be formatted differently (as in `http://localhost/articles/europe/france/finance.html`, without `?` nor `=`), the usual PHP variables won't work anymore, and only the routing system will be able to retrieve the request parameters. And you may want to add input filtering to prevent malicious code injection, which is only possible if you keep all request parameters in one clean parameter holder.

The `$sf_params` object is more powerful than just giving a getter equivalent to an array. For instance, if you only want to test the existence of a request parameter, you can simply use the `$sf_params->has()` method instead of testing the actual value with `get()`, as in Listing 4-13.

Listing 4-13 - Testing the Existence of a Request Parameter in the Template

```
<?php if ($sf_params->has('name')): ?>
  <p>Hello, <?php echo $sf_params->get('name') ?>!</p>
<?php else: ?>
  <p>Hello, John Doe!</p>
<?php endif; ?>
```

*Listing
4-19*

You may have already guessed that this can be written in a single line. As with most getter methods in symfony, both the `$request->getParameter()` method in the action and the `$sf_params->get()` method in the template (which, as a matter of fact, calls the same method on the same object) accept a second argument: the default value to be used if the request parameter is not present.

```
<p>Hello, <?php echo $sf_params->get('name', 'John Doe') ?>!</p>
```

*Listing
4-20*

Summary

In symfony, pages are composed of an action (a method in the `actions/actions.class.php` file prefixed with `execute`) and a template (a file in the `templates/` directory, usually ending with `Success.php`). They are grouped in modules, according to their function in the application. Writing templates is facilitated by helpers, which are functions provided by symfony that return HTML code. And you need to think of the URL as a part of the response, which can be formatted as needed, so you should refrain from using any direct reference to the URL in action naming or request parameter retrieval.

Once you know these basic principles, you can already write a whole web application with symfony. But it would take you way too long, since almost every task you will have to achieve during the course of the application development is facilitated one way or another by some symfony feature... which is why the book doesn't stop now.

Chapter 5

Configuring Symfony

To be simple and easy to use, symfony defines a few conventions, which should satisfy the most common requirements of standard applications without need for modification. However, using a set of simple and powerful configuration files, it is possible to customize almost everything about the way the framework and your application interact with each other. With these files, you will also be able to add specific parameters for your applications.

This chapter explains how the configuration system works:

- The symfony configuration is kept in files written in YAML, although you can always choose another format.
- Configuration files are at the project, application, and module levels in a project's directory structure.
- You can define several sets of configuration settings; in symfony, a set of configuration is called an environment.
- The values defined in the configuration files are available from the PHP code of your application.
- Additionally, symfony authorizes PHP code in YAML files and other tricks to make the configuration system even more flexible.

The Configuration System

Regardless of purpose, most web applications share a common set of characteristics. For instance, some sections can be restricted to a subset of users, or the pages can be decorated by a layout, or a form can be filled with the user input after a failed validation. A framework defines a structure for emulating these characteristics, and the developer can further tweak them by changing a configuration setting. This strategy saves a lot of development time, since many changes don't require a single line of code, even if there is a lot of code behind. It is also much more efficient, because it ensures such information can be maintained in a single and easily identifiable location.

However, this approach has two serious drawbacks:

- Developers end up writing endlessly complex XML files.
- In a PHP architecture, every request takes much longer to process.

Taking these disadvantages into account, 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 seldom need to be changed.
- Easy: Configuration files are easy to read, to modify, and to create by the developer.
- Customizable: The default configuration language is YAML, but it can be INI, XML, or whatever format the developer prefers.
- 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.

YAML Syntax and Symfony Conventions

For its configuration, symfony uses the YAML format by default, instead of more traditional INI or XML formats. YAML shows structure through indentation and is fast to write. Its advantages and basic rules were already described in Chapter 1. However, you need to keep a few conventions in mind when writing YAML files. This section introduces several of the most prominent conventions. For a complete dissertation on the topic, read the dedicated chapter on the reference guide book³¹.

First of all, never use tabs in YAML files; use spaces instead. YAML parsers can't understand files with tabs, so indent your lines with spaces (a double blank is the symfony convention for indentation), as shown in Listing 5-1.

Listing 5-1 - YAML Files Forbid Tabs

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

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

*Listing
5-1*

If your parameters are strings starting or ending with spaces, contain special characters (such as the “octothorpe” (#) or comma), or key words such as “true, false” (where a string is intended) you must enclose the value in single quotes, as shown in Listing 5-2.

Listing 5-2 - Nonstandard Strings Should Be Enclosed in Single Quotes

```
error1: This field is compulsory
error2: ' This field is compulsory '
error3: 'Don''t leave this field blank' # Single quotes must be doubled
error4: 'Enter a # symbol to define an extension number'
i18n:  'false' # if we left off the quotes here, a boolean false would be
      returned
```

*Listing
5-2*

You can define long strings in multiple lines, and also multiple-line strings, with the special string headers (> and |) plus an additional indentation. Listing 5-3 demonstrates this convention.

Listing 5-3 - Defining Long and Multiline Strings

```
# Folded style, introduced by >
# Each line break is folded to a space
# Makes YAML more readable
accomplishment: >
```

*Listing
5-3*

31. http://www.symfony-project.org/reference/1_4/en/02-YAML

```

Mark set a major league
home run record in 1998.

# Literal style, introduced by |
# All line breaks count
# Indentation doesn't appear in the resulting string
stats: |
  65 Home Runs
  0.278 Batting Average

```

To define a value as an array, enclose the elements in square brackets or use the expanded syntax with dashes, as shown in Listing 5-4.

Listing 5-4 - YAML Array Syntax

```

Listing 5-4 # Shorthand syntax for arrays
players: [ Mark McGwire, Sammy Sosa, Ken Griffey ]

# Expanded syntax for arrays
players:
  - Mark McGwire
  - Sammy Sosa
  - Ken Griffey

```

To define a value as an associative array, or hash, enclose the elements in curly brackets and always insert a spaces between the key and the value in the key: value pair, and any list items separated by commas. You can also use the expanded syntax by adding indentation and a carriage return for every new key, as shown in Listing 5-5.

Listing 5-5 - YAML Associative Array Syntax

```

Listing 5-5 # Incorrect syntax, blanks are missing after the colons and comma
mail: {webmaster:webmaster@example.com,contact:contact@example.com}

# Correct shorthand syntax for associative arrays
mail: { webmaster: webmaster@example.com, contact: contact@example.com }

# Expanded syntax for associative arrays
mail:
  webmaster: webmaster@example.com
  contact: contact@example.com

```

To give a Boolean value, you can use the `false` and `true` values, provided they are not enclosed in quotes.

Listing 5-6 - YAML Boolean Values Syntax

```

Listing 5-6 true_value: true
false_value: false

```

Don't hesitate to add comments (starting with the hash mark, `#`) and extra spaces to values to make your YAML files more readable, as shown in Listing 5-7.

Listing 5-7 - YAML Comments Syntax and Value Alignment

```

Listing 5-7 # This is a comment line
mail:
  webmaster: webmaster@example.com
  contact: contact@example.com

```

```
admin:      admin@example.com    # extra spaces allow nice alignment of
values
```

In some symfony configuration files, you will sometimes see lines that start with a hash mark (and, as such, ignored by the YAML parsers) but look like usual settings lines. This is a symfony convention: the default configuration, inherited from other YAML files located in the symfony core, is repeated in commented lines in your application configuration, for your information. If you want to change the value of such a parameter, you need to uncomment the line first, as shown in Listing 5-8.

Listing 5-8 - Default Configuration Is Shown Commented

```
# The cache is off by default
settings:
# cache: false

# If you want to change this setting, uncomment the line first
settings:
  cache: true
```

*Listing
5-8*

Symfony sometimes groups the parameter definitions into categories. All settings of a given category appear indented under the category header. Structuring long lists of key: value pairs by grouping them into categories improves the readability of the configuration. Category headers start with a dot (.). Listing 5-9 shows an example of categories.

Listing 5-9 - Category Headers Look Like Keys, But Start with a Dot

```
all:
  .general:
    tax:      19.6

  mail:
    webmaster: webmaster@example.com
```

*Listing
5-9*

In this example, `mail` is a key and `general` is only a category header. Everything works as if the category header didn't exist, as shown in Listing 5-10. The `tax` parameter is actually a direct child of the `all` key. However using categories helps symfony dealing with arrays that are beneath the `all` key.

Listing 5-10 - Category Headers Are Only There for Readability and Are Actually Ignored

```
all:
  tax:      19.6

  mail:
    webmaster: webmaster@example.com
```

*Listing
5-10*

And if you don't like YAML

YAML is just an interface to define settings to be used by PHP code, so the configuration defined in YAML files ends up being transformed into PHP. After browsing an application, check its cached configuration (in `cache/frontend/dev/config/`, for instance). You will see the PHP files corresponding to your YAML configuration. You will learn more about the configuration cache later in this chapter.

The good news is that if you don't want to use YAML files, you can still do what the configuration files do by hand, in PHP or via another format (XML, INI, and so on). Throughout this book, you will meet alternative ways to define configuration without YAML, and you will even learn to replace the `symfony` configuration handlers (in Chapter 19). If you use them wisely, these tricks will enable you to bypass configuration files or define your own configuration format.

Help, a YAML File Killed My App!

The YAML files are parsed into PHP hashes and arrays, and then the values are used in various parts of the application to modify the behavior of the view, the controller, or the model. Many times, when there is a problem in a YAML file, it is not detected until the value actually needs to be used. Moreover, the error or exception that is thrown then is usually not clearly related to the YAML configuration file.

If your application suddenly stops working after a configuration change, you should check that you didn't make any of the common mistakes of the inattentive YAML coder:

- You miss a space between a key and its value:

Listing 5-11 `key1:value1 # A space is missing after the :`

- Keys in a sequence are not indented the same way:

Listing 5-12 `all:
 key1: value1
 key2: value2 # Indentation is not the same as the other
sequence members
 key3: value3`

- There is a reserved YAML character in a key or a value, without string delimiters:

Listing 5-13 `message: tell him: go way # :, [,], { and } are reserved in
YAML
message: 'tell him: go way' # Correct syntax`

- You are modifying a commented line:

Listing 5-14 `# key: value # Will never be taken into account due to the
leading #`

- You set values with the same key name twice at the same level:

Listing 5-15 `key1: value1
key2: value2
key1: value3 # key1 is defined twice, the value is the last
one defined`

- You think that the setting takes a special type, while it is always a string, until you convert it:

```
income: 12,345 # Until you convert it, this is still a string
```

*Listing
5-16*

Overview of the Configuration Files

Configuration is distributed into files, by subject. The files contain parameter definitions, or settings. Some of these parameters can be overridden at several levels (project, application, and module); some are specific to a certain level. The next chapters will deal with the configuration files related to their main topic, and Chapter 19 will deal with advanced configuration.

Project Configuration

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

- `ProjectConfiguration.class.php`: This is the very first file included by any request or command. It contains the path to the framework files, and you can change it to use a different installation. See Chapter 19 for advanced usage of this file.
- `databases.yml`: This is where you define the access and connection settings to the database (host, login, password, database name, and so on). Chapter 8 will tell you more about it. It can also be overridden at the application level.
- `properties.ini`: This file holds a few parameters used by the command line tool, including the project name and the connection settings for distant servers. See Chapter 16 for an overview of the features using this file.
- `rsync_exclude.txt`, `rsync_include.txt`: This file specifies which directories and files must be excluded and/or included from the synchronization between servers. It is discussed in Chapter 16.
- `schema.yml`: This is the data access configuration file used by Propel and Doctrine (symfony's ORM layers). It is used to make the ORM libraries work with the symfony classes and the data of your project. `schema.yml` contains a representation of the project's relational data model. For Doctrine, this file is created in `config/doctrine/`.

These files are mostly used by external components or by the command line, or they need to be processed even before any YAML parsing program can be loaded by the framework. That's why some of them don't use the YAML format.

Application Configuration

The main part of the configuration is the application configuration. It is defined in the front controller (in the `web/` directory) for the main configuration, in YAML files located in the application `config/` directory, in `i18n/` directories for the internationalization files, and in the framework files for invisible—although useful—additional application configuration.

Front Controller Configuration

The very first application configuration is actually found in the front controller; that is the very first script executed by a request. Take a look at the default `web/index.php` in Listing 5-11.

Listing 5-11 - The Default Production Front Controller

Listing 5-17

```
<?php
require_once(dirname(__FILE__).'/../config/
ProjectConfiguration.class.php');

$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'prod',
false);
sfContext::createInstance($configuration)->dispatch();
```

After defining the name of the application (`frontend`), the environment (`prod`), and the debug mode (`false`), the application configuration is called before creating a context and dispatching. So a few useful methods are available in the application configuration class:

- `$configuration->getRootDir()`: Project root directory (normally, should remain at its default value, unless you change the file structure).
- `$configuration->getApplication()`: Application name in the project. Necessary to compute file paths.
- `$configuration->getEnvironment()`: Environment name (`prod`, `dev`, `test`, or any other project-specific environment that you define). Will determine which configuration settings are to be used. Environments are explained later in this chapter.
- `$configuration->isDebug()`: Activation of the debug mode (see Chapter 16 for details).

If you want to change one of these values, you probably need an additional front controller. The next chapter will tell you more about front controllers and how to create a new one.

Main Application Configuration

The main application configuration is stored in files located in the `myproject/apps/frontend/config/` directory:

- `app.yml`: This file should contain the application-specific configuration; that is, global variables defining business or applicative logic specific to an application, which don't need to be stored in a database. Tax rates, shipping fares, and e-mail addresses are often stored in this file. It is empty by default.
- `frontendConfiguration.class.php`: This class bootstraps the application, which means that it does all the very basic initializations to allow the application to start. This is where you can customize your directory structure or add application-specific constants (Chapter 19 provides more details). It inherits from the `sfApplicationConfiguration` class.
- `factories.yml`: Symfony defines its own class to handle the view, the request, the response, the session, and so on. If you want to use your own classes instead, this is where you can specify them. Chapter 17 provides more information.
- `filters.yml`: Filters are portions of code executed for every request. This file is where you define which filters are to be processed, and it can be overridden for each module. Chapter 6 discusses filters in more detail.
- `routing.yml`: The routing rules, which allow transforming unreadable and unbookmarkable URLs into "smart" and explicit ones, are stored in this file. For new applications, a few default rules exist. Chapter 9 is all about links and routing.
- `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 and whether caching 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. The common settings and their use are described in Chapter 19.

- `view.yml`: The structure of the default view (name of the layout, default style sheets and JavaScript files to be included, default content-type, and so on) is set in this file. Chapter 7 will tell you more about this file. These settings can be overridden for each module.



All the symfony configuration files are described in great details in the symfony reference book³².

Internationalization Configuration

Internationalized applications can display pages in several languages. This requires specific configuration. There are two configuration places for internationalization:

- The `factories.yml` of the application `config/` directory: This file defines the i18n factory and general translation options, such as the default culture for the translation, whether the translations come from files or a database, and their format.
- Translation files in the application `i18n/` directory: These are basically dictionaries, giving a translation for each of the phrases used in the application templates so that the pages show translated text when the user switches language.

Note that the activation of the i18n features is set in the `settings.yml` file. You will find more information about these features in Chapter 13.

Additional Application Configuration

A second set of configuration files is in the symfony installation directory (in `sfConfig::get('sf_symfony_lib_dir')/config/config/`) and doesn't appear in the configuration directory of your applications. The settings defined there are defaults that seldom need to be modified, or that are global to all projects. However, if you need to modify them, just create an empty file with the same name in your `myproject/apps/frontend/config/` directory, and override the settings you want to change. The settings defined in an application always have precedence over the ones defined in the framework. The following are the configuration files in the symfony installation `config/` directory:

- `autoload.yml`: This file contains the settings of the autoloading feature. This feature exempts you from requiring custom classes in your code if they are located in specific directories. It is described in detail in Chapter 19.
- `core_compile.yml`: These are lists of classes to be included to start an application. These classes are actually concatenated into an optimized PHP file without comments, which will accelerate the execution by minimizing the file access operations (one file is loaded instead of more than forty for each request). This is especially useful if you don't use a PHP accelerator. Optimization techniques are described in Chapter 18.
- `config_handlers.yml`: This is where you can add or modify the handlers used to process each configuration file. Chapter 19 provides more details.

Module Configuration

By default, a module has no specific configuration. But, if required, you can override some application-level settings for a given module. For instance, you might do this to include a specific JavaScript file for all actions of a module. You can also choose to add new parameters restricted to a specific module to preserve encapsulation.

32. http://www.symfony-project.org/reference/1_4/en/

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

- `generator.yml`: For modules generated according to a database table (scaffoldings and administrations), this file defines how the interface displays rows and fields, and which interactions are proposed to the user (filters, sorting, buttons, and so on). Chapter 14 will tell you more about it.
- `module.yml`: This file contains custom parameters specific to a module and action configuration. Chapter 6 provides more details.
- `security.yml`: This file sets access restrictions for actions. This is where you specify that a page can be viewed only by registered users or by a subset of registered users with special permissions. Chapter 6 will tell you more about it.
- `view.yml`: This file contains configuration for the views of one or all of the actions of a module. It overrides the application `view.yml` and is described in Chapter 7.

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.

Too many files?

You might be overwhelmed by the number of configuration files present in the application. But please keep the following in mind:

Most of the time, you don't need to change the configuration, since the default conventions match the most common requirements. Each configuration file is related to a particular feature, and the next chapters will detail their use one by one. When you focus on a single file, you can see clearly what it does and how it is organized. For professional web development, the default configuration is often not completely adapted. The configuration files allow for an easy modification of the symfony mechanisms without code. Imagine the amount of PHP code necessary to achieve the same amount of control. If all the configuration were located in one file, not only would the file be completely unreadable, but you could not redefine configuration at several levels (see the "Configuration Cascade" section later in this chapter).

The configuration system is one of the great strengths of symfony, because it makes symfony usable for almost every kind of web application, and not only for the ones for which the framework was originally designed.

Environments

During the course of application development, you will probably need to keep several sets of configuration in parallel. For instance, you will need to have the connection settings for your tests database available during development, and the ones for your real data available for production. To answer the need of concurrent configurations, symfony offers different environments.

What Is an Environment?

An application can run in various environments. The different environments share the same PHP code (apart from the front controller), but can have completely different configurations. For each application, symfony provides three default environments: production (`prod`), test (`test`), and development (`dev`). You're also free to add as many custom environments as you wish.

So basically, environments and configuration are synonyms. For instance, a test environment will log alerts and errors, while a `prod` environment will only log errors. Cache acceleration

is often deactivated in the `dev` environment, but activated in the `test` and `prod` environments. The `dev` and `test` environments may need test data, stored in a database distinct from the one used in the production environment. So the database configuration will be different between the two environments. All environments can live together on the same machine, although a production server generally contains only the `prod` environment.

In the `dev` environment, the logging and debugging settings are all enabled, since maintenance is more important than performance. On the contrary, the `prod` environment has settings optimized for performance by default, so the production configuration turns off many features. A good rule of thumb is to navigate in the development environment until you are satisfied with the feature you are working on, and then switch to the production environment to check its speed.

The `test` environment differs from the `dev` and `prod` environment in other ways. You interact with this environment solely through the command line for the purpose of functional testing and batch scripting. Consequently, the `test` environment is close to the production one, but it is not accessed through a web browser. It simulates the use of cookies and other HTTP specific components.

To change the environment in which you're browsing your application, just change the front controller. Until now, you have seen only the development environment, since the URLs used in the example called the development front controller:

```
http://localhost/frontend_dev.php/mymodule/index
```

*Listing
5-18*

However, if you want to see how the application reacts in production, call the production front controller instead:

```
http://localhost/index.php/mymodule/index
```

*Listing
5-19*

If your web server has `mod_rewrite` enabled, you can even use the custom symfony rewriting rules, written in `web/.htaccess`. They define the production front controller as the default execution script and allow for URLs like this:

```
http://localhost/mymodule/index
```

*Listing
5-20*

Environments and Servers

Don't mix up the notions of environment and server. In symfony, different environments are different configurations, and correspond to a front controller (the script that executes the request). Different servers correspond to different domain names in the URL.

Listing 5-21 `http://localhost/frontend_dev.php/mymodule/index`

server — environment

Usually, developers work on applications in a development server, disconnected from the Internet and where all the server and PHP configuration can be changed at will. When the time comes for releasing the application to production, the application files are transferred to the production server and made accessible to the end users.

This means that many environments are available on each server. For instance, you can run in the production environment even on your development server. However, most of the time, only the production environment should be accessible in the production server, to avoid public visibility of server configuration and security risks. To prevent accidental exposure of the non-production controllers on the production system, symfony adds a basic IP check to these front controllers, which will allow access only from localhost. If you want to have them accessible you can remove that, but think about the risk of having this accessible by anyone, as malicious users could guess the default `frontend_dev.php` and get access to a lot of sensitive information.

To add a new environment, you don't need to create a directory or to use the symfony CLI. Simply create a new front controller and change the environment name definition in it. This environment inherits all the default configuration plus the settings that are common to all environments. The next chapter will show you how to do this.

Configuration Cascade

The same setting can be defined more than once, in different places. For instance, you may want to set the mime-type of your pages to `text/html` for all of the application, except for the pages of an `rss` module, which will need a `text/xml` mime-type. Symfony gives you the ability to write the first setting in `frontend/config/view.yml` and the second in `frontend/modules/rss/config/view.yml`. The configuration system knows that a setting defined at the module level must override a setting defined at the application level.

In fact, 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/frontend/config/`)
 - The local configuration restricted to a module (in `myproject/apps/frontend/modules/mymodule/config/`)
- Environment levels:
 - Specific to one environment
 - For all environments

Of all the properties that can be customized, many are environment-dependent. Consequently, many YAML configuration files are divided by environment, plus a tail section for all environments. The result is that typical symfony configuration looks like Listing 5-12.

Listing 5-12 - The Structure of Symfony Configuration Files

```
# Production environment settings
prod:
  ...

# Development environment settings
dev:
  ...

# Test environment settings
test:
  ...

# Custom environment settings
myenv:
  ...

# Settings for all environments
all:
  ...
```

Listing
5-22

In addition, the framework itself defines default values in files that are not located in the project tree structure, but in the `sfConfig::get('sf_symfony_lib_dir')/config/config/` directory of your symfony installation. The default configuration is set in these files as shown in Listing 5-13. These settings are inherited by all applications.

Listing 5-13 - The Default Configuration, in `sfConfig::get('sf_symfony_lib_dir')/config/config/settings.yml`

```
# Default settings:
default:
  .actions:
    default_module:      default
    default_action:      index
  ...
  ...
```

Listing
5-23

These default definitions are repeated in the project, application, and module configuration files as comments, as shown in Listing 5-14, so that you know that some parameters are defined by default and that they can be modified.

Listing 5-14 - The Default Configuration, Repeated for Information, in `frontend/config/settings.yml`

```
#all:
#  .actions:
#    default_module:      default
#    default_action:      index
#    ...
#    ...
```

Listing
5-24

This means that a property can be defined several times, and the actual value results from a definition cascade. 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. A parameter definition at the module level has precedence over the same parameter definition at the application level, which has precedence over a definition at the project level. This can be wrapped up in the following priority list:

1. Module
2. Application

3. Project
4. Specific environment
5. All environments
6. Default

The Configuration Cache

Parsing YAML and dealing with the configuration cascade at runtime represent a significant overhead for each request. Symfony has a built-in configuration cache mechanism designed to speed up requests.

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

For instance, if the `app.yml` file contains this:

```
Listing 5-25 all:           # Setting for all environments
  mail:
    webmaster:      webmaster@example.com
```

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

```
Listing 5-26 <?php
sfConfig::add(array(
  'app_mail_webmaster' => 'webmaster@example.com',
));
```

As a consequence, most of the time, the YAML files aren't even parsed by the framework, which relies on the configuration cache instead. However, in the development environment, symfony will systematically compare the dates of modification of the YAML files and the cached files, and reprocess only the ones that have changed since the previous request.

This presents 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.

There is an important consequence of this mechanism. If you change the configuration in the production environment, you need to force the reparsing of all the configuration files for your modification to be taken into account. For that, you just need to clear the cache, either by deleting the content of the `cache/` directory or, more easily, by calling the `cache:clear` task:

```
Listing 5-27 $ php symfony cache:clear
```

Accessing the Configuration from Code

All the configuration files are eventually transformed into PHP, and many of the settings they contain are automatically used by the framework, without further intervention. However, you sometimes need to access some of the settings defined in the configuration files from your code (in actions, templates, custom classes, and so on). The settings defined in `settings.yml`, `app.yml`, and `module.yml` are available through a special class called `sfConfig`.

The `sfConfig` Class

You can access settings from within the application code through the `sfConfig` class. It is a registry for configuration parameters, with a simple getter class method, accessible from every part of the code:

```
// Retrieve a setting
$parameter = sfConfig::get('param_name', $default_value);
```

Listing 5-28

Note that you can also define, or override, a setting from within PHP code:

```
// Define a setting
sfConfig::set('param_name', $value);
```

Listing 5-29

The parameter name is the concatenation of several elements, separated by underscores, in this order:

- A prefix related to the configuration file name (`sf_` for `settings.yml`, `app_` for `app.yml`, `mod_` for `module.yml`)
- The parent keys (if defined), in lowercase
- The name of the key, in lowercase

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

For instance, if you need to access the values defined in the `app.yml` file shown in Listing 5-15, you will need the code shown in Listing 5-16.

Listing 5-15 - Sample app.yml Configuration

```
all:
  .general:
    tax:          19.6
  default_user:
    name:        John Doe
  mail:
    webmaster:   webmaster@example.com
    contact:     contact@example.com
dev:
  mail:
    webmaster:   dummy@example.com
    contact:     dummy@example.com
```

Listing 5-30

Listing 5-16 - Accessing Configuration Settings in PHP in the dev Environment

```
echo sfConfig::get('app_tax'); // Remember that category headers are
ignored
=> '19.6'
echo sfConfig::get('app_default_user_name');
```

Listing 5-31

```
=> 'John Doe'
echo sfConfig::get('app_mail_webmaster');
=> 'dummy@example.com'
echo sfConfig::get('app_mail_contact');
=> 'dummy@example.com'
```

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

On that account, the `settings.yml` file, where you can set the framework settings for an application, is the equivalent to a list of `sfConfig::set()` calls. Listing 5-17 is interpreted as shown in Listing 5-18.

Listing 5-17 - Extract of `settings.yml`

```
Listing 5-32 all:
  .settings:
    csrf_secret:      FooBar
    escaping_strategy: true
    escaping_method:   ESC_SPECIALCHARS
```

Listing 5-18 - What Symfony Does When Parsing `settings.yml`

```
Listing 5-33 sfConfig::add(array(
  'sf_csrf_secret' => 'FooBar',
  'sf_escaping_strategy' => true,
  'sf_escaping_method' => 'ESC_SPECIALCHARS',
));
```

Refer to Chapter 19 for the meanings of the settings found in the `settings.yml` file.

Custom Application Settings and `app.yml`

Most of the settings related to the features of an application should be stored in the `app.yml` file, located in the `myproject/apps/frontend/config/` directory. This file is environment-dependent and empty by default. Put in every setting that you want to be easily changed, and use the `sfConfig` class to access these settings from your code. Listing 5-19 shows an example.

Listing 5-19 - Sample `app.yml` to Define Credit Card Operators Accepted for a Given Site

```
Listing 5-34 all:
  creditcards:
    fake:          false
    visa:          true
    americanexpress: true

dev:
  creditcards:
    fake:          true
```

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

```
Listing 5-35 sfConfig::get('app_creditcards_fake');
```



When you should require an PHP array directly beneath the `all` key you need to use a category header, otherwise symfony will make the values separately available as shown above.

```

all:
  .array:
    creditcards:
      fake:          false
      visa:          true
      americanexpress: true

[php]
print_r(sfConfig::get('app_creditcards'));

Array(
  [fake] => false
  [visa] => true
  [americanexpress] => true
)

```

Listing 5-36

TIP Each time you are tempted to define a constant or a setting in one of your scripts, think about if it would be better located in the `app.yml` file. This is a very convenient place to store all application settings.

When your need for custom parameters becomes hard to handle with the `app.yml` syntax, you may need to define a syntax of your own. In that case, you can store the configuration in a new file, interpreted by a new configuration handler. Refer to Chapter 19 for more information about configuration handlers.

Tips for Getting More from Configuration Files

There are a few last tricks to learn before writing your own YAML files. They will allow you to avoid configuration duplication and to deal with your own YAML formats.

Using Constants in YAML Configuration Files

Some configuration settings rely on the value of other settings. To avoid setting the same value twice, symfony supports constants in YAML files. On encountering a setting name (one that can be accessed by `sfConfig::get()`) in capital letters enclosed in % signs, the configuration handlers replace them with their current value. See Listing 5-20 for an example.

Listing 5-20 - Using Constants in YAML Files, Example from `autoload.yml`

```

autoload:
  symfony:
    name:          symfony
    path:          %SF_SYMFONY_LIB_DIR%
    recursive:    true
    exclude:      [vendor]

```

Listing 5-37

The `path` parameter will take the value returned by `sfConfig::get('sf_symfony_lib_dir')`. If you want one configuration file to rely on another, you need to make sure that the file you rely on is already parsed (look in the symfony source to find out the order in which the configuration files are parsed). `app.yml` is one of the last files parsed, so you may rely on others in it.

All the available constants are described in the symfony reference book³³.

33. http://www.symfony-project.org/reference/1_4/en/

Using Scriptable Configuration

It may happen that your configuration relies on external parameters (such as a database or another configuration file). To deal with these particular cases, the symfony configuration files are parsed as PHP files before being passed to the YAML parser. It means that you can put PHP code in YAML files, as in Listing 5-21.

Listing 5-21 - YAML Files Can Contain PHP

```
Listing 5-38 all:
  translation:
    format:  <?php echo (sfConfig::get('sf_i18n') === true ? 'xlf' :
null)."\\n" ?>
```

But be aware that the configuration is parsed very early in the life of a request, so you will not have any symfony built-in methods or functions to help you.

Also, as the `echo` language construct does not add a carriage return by default, you need to add a “`\n`” or use the `echoln` helper to keep the YAML format valid.

```
Listing 5-39 all:
  translation:
    format:  <?php echoln(sfConfig::get('sf_i18n') == true ? 'xlf' :
'none') ?>
```



In the production environment, the configuration is cached, so the configuration files are parsed (and executed) only once after the cache is cleared.

Browsing Your Own YAML File

Whenever you want to read a YAML file directly, you can use the `sfYaml` class. It is a YAML parser that can turn a YAML file into a PHP associative array. Listing 5-22 presents a sample YAML file, and Listing 5-23 shows you how to parse it.

Listing 5-22 - Sample test.yml File

```
Listing 5-40 house:
  family:
    name:      Doe
    parents:   [John, Jane]
    children:  [Paul, Mark, Simone]
  address:
    number:    34
    street:    Main Street
    city:      Nowheretown
    zipcode:   12345
```

Listing 5-23 - Using the sfYaml Class to Turn a YAML File into an Associative Array

```
Listing 5-41 $test = sfYaml::load('/path/to/test.yml');
print_r($test);
```

```
Array(
  [house] => Array(
    [family] => Array(
      [name] => Doe
      [parents] => Array(
```

```
[0] => John
[1] => Jane
)
[children] => Array(
    [0] => Paul
    [1] => Mark
    [2] => Simone
)
)
[address] => Array(
    [number] => 34
    [street] => Main Street
    [city] => Nowheretown
    [zipcode] => 12345
)
)
)
```

Summary

The symfony configuration system uses the YAML language to be simple and readable. The ability to deal with multiple environments and to set parameters through a definition cascade offers versatility to the developer. Some of the configuration can be accessed from within the code via the `sfConfig` object, especially the application settings stored in the `app.yml` file.

Yes, symfony does have a lot of configuration files, but this approach makes it more adaptable. Remember that you don't need to bother with them unless your application requires a high level of customization.

Chapter 6

Inside The Controller Layer

In `symfony`, the controller layer, which contains the code linking the business logic and the presentation, is split into several components that you use for different purposes:

- The front controller is the unique entry point to the application. It loads the configuration and determines the action to execute.
- Actions contain the applicative logic. They check the integrity of the request and prepare the data needed by the presentation layer.
- The request, response, and session objects give access to the request parameters, the response headers, and the persistent user data. They are used very often in the controller layer.
- Filters are portions of code executed for every request, before or after the action. For example, the security and validation filters are commonly used in web applications. You can extend the framework by creating your own filters.

This chapter describes all these components, but don't be intimidated by their number. For a basic page, you will probably need to write only a few lines in the action class, and that's all. The other controller components will be of use only in specific situations.

The 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.

When the front controller receives a request, it uses the routing system to match an action name and a module name with the URL typed (or clicked) by the user. For instance, the following request URL calls the `index.php` script (that's the front controller) and will be understood as a call to the action `myAction` of the module `mymodule`:

Listing 6-1 `http://localhost/index.php/mymodule/myAction`

If you are not interested in `symfony`'s internals, that's all that you need to know about the front controller. It is an indispensable component of the `symfony` MVC architecture, but you will seldom need to change it. So you can jump to the next section unless you really want to know about the guts of the front controller.

The Front Controller's Job in Detail

The front controller does the dispatching of the request, but that means a little more than just determining the action to execute. In fact, it executes the code that is common to all actions, including the following:

1. Load the project configuration class and the `symfony` libraries.

2. Create an application configuration and a symfony context.
3. Load and initiate the core framework classes.
4. Load the configuration.
5. Decode the request URL to determine the action to execute and the request parameters.
6. If the action does not exist, redirect to the 404 error action.
7. Activate filters (for instance, if the request needs authentication).
8. Execute the filters, first pass.
9. Execute the action and render the view.
10. Execute the filters, second pass.
11. Output the response.

The Default Front Controller

The default front controller, called `index.php` and located in the `web/` directory of the project, is a simple PHP file, as shown in Listing 6-1.

Listing 6-1 - The Default Production Front Controller

```
<?php
require_once(dirname(__FILE__).'/../config/
ProjectConfiguration.class.php');

$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'prod',
false);
sfContext::createInstance($configuration)->dispatch();
```

*Listing
6-2*

The front controller creates an application configuration instance, which takes care of steps 2 through 4. The call to the `dispatch()` method of the `sfController` object (which is the core controller object of the symfony MVC architecture) dispatches the request, taking care of steps 5 through 7. The last steps are handled by the filter chain, as explained later in this chapter.

Calling Another Front Controller to Switch the Environment

One front controller exists per environment. As a matter of fact, it is the very existence of a front controller that defines an environment. The environment is defined by the second argument you pass to the `ProjectConfiguration::getApplicationConfiguration()` method call.

To change the environment in which you're browsing your application, just choose another front controller. The default front controllers available when you create a new application with the `generate:app` task are `index.php` for the production environment and `frontend_dev.php` for the development environment (provided that your application is called `frontend`). The default `mod_rewrite` configuration will use `index.php` when the URL doesn't contain a front controller script name. So both of these URLs display the same page (`mymodule/index`) in the production environment:

`http://localhost/index.php/mymodule/index`
`http://localhost/mymodule/index`

*Listing
6-3*

and this URL displays that same page in the development environment:

`http://localhost/frontend_dev.php/mymodule/index`

*Listing
6-4*

Creating a new environment is as easy as creating a new front controller. For instance, you may need a staging environment to allow your customers to test the application before going to production. To create this staging environment, just copy `web/frontend_dev.php` into `web/frontend_staging.php`, and change the value of the second argument of the `ProjectConfiguration::getApplicationConfiguration()` call to `staging`. Now, in all the configuration files, you can add a new `staging:` section to set specific values for this environment, as shown in Listing 6-2.

Listing 6-2 - Sample app.yml with Specific Settings for the Staging Environment

```
Listing 6-5
staging:
  mail:
    webmaster: dummy@mysite.com
    contact: dummy@mysite.com
all:
  mail:
    webmaster: webmaster@mysite.com
    contact: contact@mysite.com
```

If you want to see how the application reacts in this new environment, call the related front controller:

```
Listing 6-6 http://localhost/frontend_staging.php/mymodule/index
```

Actions

The actions are the heart of an application, because they contain all the application's logic. They call the model and define variables for the view. When you make a web request in a symfony application, the URL defines an action and the request parameters.

The Action Class

Actions are methods named `executeActionName` of a class named `moduleNameActions` inheriting from the `sfActions` class, and grouped by modules. The action class of a module is stored in an `actions.class.php` file, in the module's `actions/` directory.

Listing 6-3 shows an example of an `actions.class.php` file with only an `index` action for the whole `mymodule` module.

Listing 6-3 - Sample Action Class, in apps/frontend/modules/mymodule/actions/actions.class.php

```
Listing 6-7
class mymoduleActions extends sfActions
{
  public function executeIndex($request)
  {
    // ...
  }
}
```



Even if method names are not case-sensitive in PHP, they are in symfony. So don't forget that the action methods must start with a lowercase `execute`, followed by the exact action name with the first letter capitalized.

In order to request an action, you need to call the front controller script with the module name and action name as parameters. By default, this is done by appending the couple

`module_name/action_name` to the script. This means that the action defined in Listing 6-4 can be called by this URL:

`http://localhost/index.php/mymodule/index`

Listing 6-8

Adding more actions just means adding more `execute` methods to the `sfActions` object, as shown in Listing 6-4.

Listing 6-4 - Action Class with Two Actions, in frontend/modules/mymodule/actions/actions.class.php

```
class mymoduleActions extends sfActions
{
    public function executeIndex($request)
    {
        // ...
    }

    public function executeList($request)
    {
        // ...
    }
}
```

Listing 6-9

If the size of an action class grows too much, you probably need to do some refactoring and move some code to the model layer. Actions should often be kept short (not more than a few lines), and all the business logic should usually be in the model.

Still, the number of actions in a module can be important enough to lead you to split it in two modules.

Symfony coding standards

In the code examples given in this book, you probably noticed that the opening and closing curly braces (`{` and `}`) occupy one line each. This standard makes the code easier to read.

Among the other coding standards of the framework, indentation is always done by two blank spaces; tabs are not used. This is because tabs have a different space value according to the text editor you use, and because code with mixed tab and blank indentation is impossible to read.

Core and generated symfony PHP files do not end with the usual `?>` closing tag. This is because it is not really needed, and because it can create problems in the output if you ever have blanks after this tag.

And if you really pay attention, you will see that a line never ends with a blank space in symfony. The reason, this time, is more prosaic: lines ending with blanks look ugly in Fabien's text editor.

Alternative Action Class Syntax

An alternative action syntax is available to dispatch the actions in separate files, one file per action. In this case, each action class extends `sfAction` (instead of `sfActions`) and is named `actionNameAction`. The actual action method is simply named `execute`. The file name is the same as the class name. This means that the equivalent of Listing 6-4 can be written with the two files shown in Listings 6-5 and 6-6.

Listing 6-5 - Single Action File, in frontend/modules/mymodule/actions/indexAction.class.php

Listing 6-10

```
class indexAction extends sfAction
{
    public function execute($request)
    {
        // ...
    }
}
```

Listing 6-6 - Single Action File, in frontend/modules/mymodule/actions/listAction.class.php

Listing 6-11

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

Retrieving Information in the Action

The action class offers a way to access controller-related information and the core symfony objects. Listing 6-7 demonstrates how to use them.

Listing 6-7 - sfActions Common Methods

Listing 6-12

```
class mymoduleActions extends sfActions
{
    public function executeIndex(sfWebRequest $request)
    {
        // Retrieving request parameters
        $password = $request->getParameter('password');

        // Retrieving controller information
        $moduleName = $this->getModuleName();
        $actionName = $this->getActionName();

        // Retrieving framework core objects
        $userSession = $this->getUser();
        $response = $this->getResponse();
        $controller = $this->getController();
        $context = $this->getContext();

        // Setting action variables to pass information to the template
        $this->setVar('foo', 'bar');
        $this->foo = 'bar';           // Shorter version
    }
}
```

The context singleton

You already saw, in the front controller, a call to `sfContext::createInstance()`. In an action, the `getContext()` method returns the same singleton. It is a very useful object that stores a reference to all the symfony core objects related to a given request, and offers an accessor for each of them:

```
sfController: The controller object (->getController())
sfRequest: The request object (->getRequest())
sfResponse: The response object (->getResponse())
sfUser: The user session object (->getUser())
sfRouting: The routing object (->getRouting())
sfMailer: The mailer object (->getMailer())
sfI18N: The internationalization object (->getI18N())
sfLogger: The logger object (->getLogger())
sfDatabaseConnection: The database connection (->getDatabaseConnection())
```

All these core objects are available through the `sfContext::getInstance()` singleton from any part of the code. However, it's a really bad practice because this will create some hard dependencies making your code really hard to test, reuse and maintain. You will learn in this book how to avoid the usage of `sfContext::getInstance()`.

Action Termination

Various behaviors are possible at the conclusion of an action's execution. The value returned by the action method determines how the view will be rendered. Constants of the `sfView` class are used to specify which template is to be used to display the result of the action.

If there is a default view to call (this is the most common case), the action should end as follows:

```
return sfView::SUCCESS;
```

Listing 6-13

Symfony will then look for a template called `actionNameSuccess.php`. This is defined as the default action behavior, so if you omit the `return` statement in an action method, symfony will also look for an `actionNameSuccess.php` template. Empty actions will also trigger that behavior. See Listing 6-8 for examples of successful action termination.

Listing 6-8 - Actions That Will Call the `indexSuccess.php` and `listSuccess.php` Templates

```
public function executeIndex()
{
    return sfView::SUCCESS;
}

public function executeList()
{
}
```

Listing 6-14

If there is an error view to call, the action should end like this:

```
return sfView::ERROR;
```

Listing 6-15

Symfony will then look for a template called `actionNameError.php`.

To call a custom view, use this ending:

Listing 6-16 `return 'MyResult';`

Symfony will then look for a template called `actionNameMyResult.php`.

If there is no view to call—for instance, in the case of an action executed in a batch process—the action should end as follows:

Listing 6-17 `return sfView::NONE;`

No template will be executed in that case. It means that you can bypass completely the view layer and set the response HTML code directly from an action. As shown in Listing 6-9, symfony provides a specific `renderText()` method for this case. This can be useful when you need extreme responsiveness of the action, such as for Ajax interactions, which will be discussed in Chapter 11.

Listing 6-9 - Bypassing the View by Echoing the Response and Returning sfView::NONE

Listing 6-18 `public function executeIndex()
{
 $this->getResponse()->setContent("<html><body>Hello,
World!</body></html>");

 return sfView::NONE;
}

// Is equivalent to
public function executeIndex()
{
 return $this->renderText("<html><body>Hello, World!</body></html>");
}`

In some cases, you need to send an empty response but with some headers defined in it (especially the X-JSON header). Define the headers via the `sfResponse` object, discussed in the next chapter, and return the `sfView::HEADER_ONLY` constant, as shown in Listing 6-10.

Listing 6-10 - Escaping View Rendering and Sending Only Headers

Listing 6-19 `public function executeRefresh()
{
 $output = '<"title","My basic letter"],["name","Mr Brown">';
 $this->getResponse()->setHttpHeader("X-JSON", '('.$.output.')');

 return sfView::HEADER_ONLY;
}`

If the action must be rendered by a specific template, ignore the `return` statement and use the `setTemplate()` method instead.

Listing 6-20 `public function executeIndex()
{
 $this->setTemplate('myCustomTemplate');
}`

With this code, symfony will look for a `myCustomTemplateSuccess.php` file, instead of `indexSuccess.php`.

Skipping to Another Action

In some cases, the action execution ends by requesting a new action execution. For instance, an action handling a form submission in a POST request usually redirects to another action after updating the database.

The action class provides two methods to execute another action:

- If the action forwards the call to another action:

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

Listing 6-21

- If the action results in a web redirection:

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

Listing 6-22



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. They throw an `sfStopException` to stop the execution of the action; this exception is later caught by symfony and simply ignored.

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 to the user. As far as the user is concerned, the displayed URL is the same as the one requested. In contrast, 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 is called from a submitted form with `method="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 by displaying the form and not an alert asking the user if he wants to resubmit a POST request.

There is a special kind of forward that is used very commonly. The `forward404()` method forwards to a "page not found" action. This method is often called when a parameter necessary to the action execution is not present in the request (thus detecting a wrongly typed URL). Listing 6-11 shows an example of a `show` action expecting an `id` parameter.

Listing 6-11 - Use of the `forward404()` Method

```
public function executeShow(sfWebRequest $request)
{
    // Doctrine
    $article =
Doctrine::getTable('Article')->find($request->getParameter('id'));

    // Propel
    $article = ArticlePeer::retrieveByPK($request->getParameter('id'));

    if (!$article)
    {
        $this->forward404();
    }
}
```

Listing 6-23



If you are looking for the error 404 action and template, you will find them in the `$sf_symfony_ lib_dir/controller/default/` directory. You can customize this page by adding a new `default` module to your application, overriding the one located in

the framework, and by defining an `error404` action and an `error404Success` template inside. Alternatively, you can set the `error_404_module` and `error_404_action` constants in the `settings.yml` file to use an existing action.

Experience shows that, most of the time, an action makes a redirect or a forward after testing something, such as in Listing 6-12. That's why the `sfActions` class has a few more methods, named `forwardIf()`, `forwardUnless()`, `forward404If()`, `forward404Unless()`, `redirectIf()`, and `redirectUnless()`. These methods simply take an additional parameter representing a condition that triggers the execution if tested true (for the `xxxIf()` methods) or false (for the `xxxUnless()` methods), as illustrated in Listing 6-12.

Listing 6-12 - Use of the `forward404If()` Method

Listing 6-24

```
// This action is equivalent to the one shown in Listing 6-11
public function executeShow(sfWebRequest $request)
{
    $article =
Doctrine::getTable('Article')->find($request->getParameter('id'));
    $this->forward404If(!$article);
}

// So is this one
public function executeShow(sfWebRequest $request)
{
    $article =
Doctrine::getTable('Article')->find($request->getParameter('id'));
    $this->forward404Unless($article);
}
```

Using these methods will not only keep your code short, but it will also make it more readable.



When the action calls `forward404()` or its fellow methods, symfony throws an `sfError404Exception` that manages the 404 response. This means that if you need to display a 404 message from somewhere where you don't want to access the controller, you can just throw a similar exception.

Repeating Code for Several Actions of a Module

The convention to name actions `executeActionName()` (in the case of an `sfActions` class) or `execute()` (in the case of an `sfAction` class) guarantees that symfony will find the action method. It gives you the ability to add other methods of your own that will not be considered as actions, as long as they don't start with `execute`.

There is another useful convention for when you need to repeat several statements in each action before the actual action execution. You can then extract them into the `preExecute()` method of your action class. You can probably guess how to repeat statements after every action is executed: wrap them in a `postExecute()` method. The syntax of these methods is shown in Listing 6-13.

Listing 6-13 - Using `preExecute()`, `postExecute()`, and Custom Methods in an Action Class

Listing 6-25

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

```

    // The code inserted here is executed at the beginning of each action
call
    ...
}

public function executeIndex($request)
{
    ...
}

public function executeList($request)
{
    ...
    $this->myCustomMethod(); // Methods of the action class are accessible
}

public function postExecute()
{
    // The code inserted here is executed at the end of each action call
    ...
}

protected function myCustomMethod()
{
    // You can also add your own methods, as long as they don't start with
"execute"
    // In that case, it's better to declare them as protected or private
    ...
}
}

```



As the pre/post execute methods are called for **each** actions of the current module, be sure you really need the execute this code for **all** your actions, to avoid unwanted side-effects.

Accessing the Request

The first argument passed to any action method is the request object, called `sfWebRequest` in symfony. You're already familiar with the `getParameter('myparam')` method, used to retrieve the value of a request parameter by its name. Table 6-1 lists the most useful `sfWebRequest` methods.

Table 6-1 - Methods of the `sfWebRequest` Object

Name	Function	Sample Output
Request Information		
<code>isMethod(\$method)</code>	Is it a post or a get?	true or false
<code>getMethod()</code>	Request method name	'POST'
<code>getHttpHeader('Server')</code>	Value of a given HTTP header	'Apache/2.0.59 (Unix) DAV/2 PHP/5.1.6'
<code>getCookie('foo')</code>	Value of a named cookie	'bar'

Name	Function	Sample Output
<code>isXmlHttpRequest()*</code>	Is it an Ajax request?	<code>true</code>
<code>isSecure()</code>	Is it an SSL request?	<code>true</code>
Request Parameters		
<code>hasParameter('foo')</code>	Is a parameter present in the request?	<code>true</code>
<code>getParameter('foo')</code>	Value of a named parameter	<code>'bar'</code>
<code>getParameterHolder()->getAll()</code>	Array of all request parameters	
URI-Related Information		
<code>getUri()</code>	Full URI	<code>'http://localhost/frontend_dev.php/mymodule/myaction'</code>
<code>getPathInfo()</code>	Path info	<code>'/mymodule/myaction'</code>
<code>getReferer()**</code>	Referrer	<code>'http://localhost/frontend_dev.php/'</code>
<code>getHost()</code>	Host name	<code>'localhost'</code>
<code>getScriptName()</code>	Front controller path and name	<code>'frontend_dev.php'</code>
Client Browser Information		
<code>getLanguages()</code>	Array of accepted languages	<code>Array([0] => fr [1] => fr_FR [2] => en_US [3] => en)</code>
<code>getCharsets()</code>	Array of accepted charsets	<code>Array([0] => ISO-8859-1 [1] => UTF-8 [2] => *)</code>
<code>getAcceptableContentTypes()</code>	Array of accepted content types	<code>Array([0] => text/xml [1] => text/html)</code>

* Works with prototype, Prototype, Mootools, and jQuery

** Sometimes blocked by proxies

You don't have to worry about whether your server supports the `$_SERVER` or the `$_ENV` variables, or about default values or server-compatibility issues—the `sfWebRequest` methods do it all for you. Besides, their names are so evident that you will no longer need to browse the PHP documentation to find out how to get information from the request.

User Session

Symfony automatically manages user sessions and is able to keep persistent data between requests for users. It uses the built-in PHP session-handling mechanisms and enhances them to make them more configurable and easier to use.

Accessing the User Session

The session object for the current user is accessed in the action with the `getUser()` method and is an instance of the `sfUser` class. This class contains a parameter holder that allows you

to store any user attribute in it. This data will be available to other requests until the end of the user session, as shown in Listing 6-14. User attributes can store any type of data (strings, arrays, and associative arrays). They can be set for every individual user, even if that user is not identified.

Listing 6-14 - The sfUser Object Can Hold Custom User Attributes Existing Across Requests

```
class mymoduleActions extends sfActions
{
    public function executeFirstPage($request)
    {
        $nickname = $request->getParameter('nickname');

        // Store data in the user session
        $this->getUser()->setAttribute('nickname', $nickname);
    }

    public function executeSecondPage()
    {
        // Retrieve data from the user session with a default value
        $nickname = $this->getUser()->getAttribute('nickname', 'Anonymous
Coward');
    }
}
```

*Listing
6-26*



You can store objects in the user session, but it is strongly discouraged. This is because the session object is serialized between requests. When the session is deserialized, the class of the stored objects must already be loaded, and that's not always the case. In addition, there can be "stalled" objects if you store Propel or Doctrine objects.

Like many getters in symfony, the `getAttribute()` method accepts a second argument, specifying the default value to be used when the attribute is not defined. To check whether an attribute has been defined for a user, use the `hasAttribute()` method. The attributes are stored in a parameter holder that can be accessed by the `getAttributeHolder()` method. It allows for easy cleanup of the user attributes with the usual parameter holder methods, as shown in Listing 6-15.

Listing 6-15 - Removing Data from the User Session

```
class mymoduleActions extends sfActions
{
    public function executeRemoveNickname()
    {
        $this->getUser()->getAttributeHolder()->remove('nickname');
    }

    public function executeCleanup()
    {
        $this->getUser()->getAttributeHolder()->clear();
    }
}
```

*Listing
6-27*

The user session attributes are also available in the templates by default via the `$sf_user` variable, which stores the current `sfUser` object, as shown in Listing 6-16.

Listing 6-16 - Templates Also Have Access to the User Session Attributes

*Listing
6-28*

```
<p>
    Hello, <?php echo $sf_user->getAttribute('nickname') ?>
</p>
```

Flash Attributes

A recurrent problem with user attributes is the cleaning of the user session once the attribute is not needed anymore. For instance, you may want to display a confirmation after updating data via a form. As the form-handling action makes a redirect, the only way to pass information from this action to the action it redirects to is to store the information in the user session. But once the confirmation message is displayed, you need to clear the attribute; otherwise, it will remain in the session until it expires.

The flash attribute is an ephemeral attribute that you can define and forget, knowing that it will disappear after the very next request and leave the user session clean for the future. In your action, define the flash attribute like this:

Listing 6-29 `$this->getUser()->setFlash('notice', $value);`

The template will be rendered and 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 like this:

Listing 6-30 `$value = $this->getUser()->getFlash('notice');`

Then forget about it. After delivering this second page, the `notice` flash attribute will be flushed. 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_user` object:

Listing 6-31 `<?php if ($sf_user->hasFlash('notice')): ?>
 <?php echo $sf_user->getFlash('notice') ?>
<?php endif; ?>`

or just:

Listing 6-32 `<?php echo $sf_user->getFlash('notice') ?>`

Flash attributes are a clean way of passing information to the very next request.

Session Management

Symfony's session-handling feature completely masks the client and server storage of the session IDs to the developer. However, if you want to modify the default behaviors of the session-management mechanisms, it is still possible. This is mostly for advanced users.

On the client side, sessions are handled by cookies. The symfony session cookie is called `symfony`, but you can change its name by editing the `factories.yml` configuration file, as shown in Listing 6-17.

Listing 6-17 - Changing the Session Cookie Name, in apps/frontend/config/factories.yml

Listing 6-33 `all:
 storage:
 class: sfSessionStorage
 param:
 session_name: my_cookie_name`



The session is started (with the PHP function `session_start()`) only if the `auto_start` parameter is set to true in `factories.yml` (which is the case by default). If you want to start the user session manually, disable this setting of the storage factory.

Symfony's session handling is based on PHP sessions. This means that if you want the client-side management of sessions to be handled by URL parameters instead of cookies, you just need to change the `use_trans_sid` setting in your `php.ini`. Be aware that this is not recommended.

```
session.use_trans_sid = 1
```

Listing 6-34

On the server side, symfony stores user sessions in files by default. You can store them in your database by changing the value of the `class` parameter in `factories.yml`, as shown in Listing 6-18.

Listing 6-18 - Changing the Server Session Storage, in `apps/frontend/config/factories.yml`

```
all:
  storage:
    class: sfMySQLSessionStorage
    param:
      db_table: session          # Name of the table storing the
sessions
      database: propel           # Name of the database connection
to use
      # Optional parameters
      db_id_col: sess_id         # Name of the column storing the
session id
      db_data_col: sess_data     # Name of the column storing the
session data
      db_time_col: sess_time     # Name of the column storing the
session timestamp
```

Listing 6-35

The `database` setting defines the database connection to be used. Symfony will then use `databases.yml` (see Chapter 8) to determine the connection settings (host, database name, user, and password) for this connection.

The available session storage classes are `sfCacheSessionStorage`, `sfMySQLSessionStorage`, `sfMySQLiSessionStorage`, `sfPostgreSQLSessionStorage`, and `sfPDOStorage`; the latter is preferred. To disable session storage completely, you can use the `sfNoStorage` class.

Session expiration occurs automatically after 30 minutes. This default setting can be modified for each environment in the same `factories.yml` configuration file, but this time in the `user` factory, as shown in Listing 6-19.

Listing 6-19 - Changing Session Lifetime, in `apps/frontend/config/factories.yml`

```
all:
  user:
    class: myUser
    param:
      timeout: 1800          # Session lifetime in seconds
```

Listing 6-36

To learn more about factories, refer to Chapter 19.

Action Security

The ability to execute an action can be restricted to users with certain privileges. The tools provided by symfony for this purpose allow the creation of secure applications, where users need to be authenticated before accessing some features or parts of the application. Securing an application requires two steps: declaring the security requirements for each action and logging in users with privileges so that they can access these secure actions.

Access Restriction

Before being executed, every action passes by a special filter that checks if the current user has the privileges to access the requested action. In symfony, privileges are composed of two parts:

- Secure actions require users to be authenticated.
- Credentials are named security privileges that allow organizing security by group.

Restricting access to an action is simply made by creating and editing a YAML configuration file called `security.yml` in the module `config/` directory. In this file, you can specify the security requirements that users must fulfill for each action or for all actions. Listing 6-20 shows a sample `security.yml`.

Listing 6-20 - Setting Access Restrictions, in apps/frontend/modules/mymodule/config/security.yml

```
Listing 6-37
read:
    is_secure: false      # All users can request the read action

update:
    is_secure: true       # The update action is only for authenticated
    users

delete:
    is_secure: true       # Only for authenticated users
    credentials: admin    # With the admin credential

all:
    is_secure: false      # false is the default value anyway
```

Actions are not secure by default, so when there is no `security.yml` or no mention of an action in it, actions are accessible by everyone. If there is a `security.yml`, symfony looks for the name of the requested action and, if it exists, checks the fulfillment of the security requirements. What happens when a user tries to access a restricted action depends on his credentials:

- If the user is authenticated and has the proper credentials, the action is executed.
- If the user is not identified, he will be redirected to the default login action.
- If the user is identified but doesn't have the proper credentials, he will be redirected to the default secure action, shown in Figure 6-1.

The default login and secure pages are pretty simple, and you will probably want to customize them. You can configure which actions are to be called in case of insufficient privileges in the application `settings.yml` by changing the value of the properties shown in Listing 6-21.

Figure 6-1 - The default secure action page

Listing 6-21 - Default Security Actions Are Defined in apps/frontend/config/settings.yml

```
all:
  .actions:
    login_module: default
    login_action: login

    secure_module: default
    secure_action: secure
```

Listing 6-38

Granting Access

To get access to restricted actions, users need to be authenticated and/or to have certain credentials. You can extend a user's privileges by calling methods of the `sfUser` object. The authenticated status of the user is set by the `setAuthenticated()` method and can be checked with `isAuthenticated()`. Listing 6-22 shows a simple example of user authentication.

Listing 6-22 - Setting the Authenticated Status of a User

```
class myAccountActions extends sfActions
{
  public function executeLogin($request)
  {
    if ($request->getParameter('login') === 'foobar')
    {
      $this->getUser()->setAuthenticated(true);
    }
  }

  public function executeLogout()
  {
    $this->getUser()->setAuthenticated(false);
  }
}
```

Listing 6-39

Credentials are a bit more complex to deal with, since you can check, add, remove, and clear credentials. Listing 6-23 describes the credential methods of the `sfUser` class.

Listing 6-23 - Dealing with User Credentials in an Action

```
Listing 6-40
class myAccountActions extends sfActions
{
    public function executeDoThingsWithCredentials()
    {
        $user = $this->getUser();

        // Add one or more credentials
        $user->addCredential('foo');
        $user->addCredentials('foo', 'bar');

        // Check if the user has a credential
        echo $user->hasCredential('foo');                      => true

        // Check if the user has both credentials
        echo $user->hasCredential(array('foo', 'bar'));        => true

        // Check if the user has one of the credentials
        echo $user->hasCredential(array('foo', 'bar'), false); => true

        // Remove a credential
        $user->removeCredential('foo');
        echo $user->hasCredential('foo');                      => false

        // Remove all credentials (useful in the logout process)
        $user->clearCredentials();
        echo $user->hasCredential('bar');                      => false
    }
}
```

If a user has the `foo` credential, that user will be able to access the actions for which the `security.yml` requires that credential. Credentials can also be used to display only authorized content in a template, as shown in Listing 6-24.

Listing 6-24 - Dealing with User Credentials in a Template

```
Listing 6-41
<ul>
    <li><?php echo link_to('section1', 'content/section1') ?></li>
    <li><?php echo link_to('section2', 'content/section2') ?></li>
    <?php if ($sf_user->hasCredential('section3')): ?>
        <li><?php echo link_to('section3', 'content/section3') ?></li>
    <?php endif; ?>
</ul>
```

As for the authenticated status, credentials are often given to users during the login process. This is why the `sfUser` object is often extended to add login and logout methods, in order to set the security status of users in a central place.



Among the symfony plug-ins, the `sfGuardPlugin`³⁴ and `sfDoctrineGuardPlugin`³⁵ extend the session class to make login and logout easy. Refer to Chapter 17 for more information.

34. <http://www.symfony-project.org/plugins/sfGuardPlugin>

35. <http://www.symfony-project.org/plugins/sfDoctrineGuardPlugin>

Complex Credentials

The YAML syntax used in the `security.yml` file allows you to restrict access to users having a combination of credentials, using either AND-type or OR-type associations. With such a combination, you can build a complex workflow and user privilege management system—for instance, a content management system (CMS) back-office accessible only to users with the `admin` credential, where articles can be edited only by users with the `editor` credential and published only by the ones with the `publisher` credential. Listing 6-25 shows this example.

Listing 6-25 - Credentials Combination Syntax

```
editArticle:
    credentials: [ admin, editor ]           # admin AND editor

publishArticle:
    credentials: [ admin, publisher ]        # admin AND publisher

userManagement:
    credentials: [[ admin, superuser ]]       # admin OR superuser
```

*Listing
6-42*

Each time you add a new level of square brackets, the logic swaps between AND and OR. So you can create very complex credential combinations, such as this:

```
credentials: [[root, [supplier, [owner, quasiowner]], accounts]]
# root OR (supplier AND (owner OR quasiowner)) OR accounts
```

*Listing
6-43*

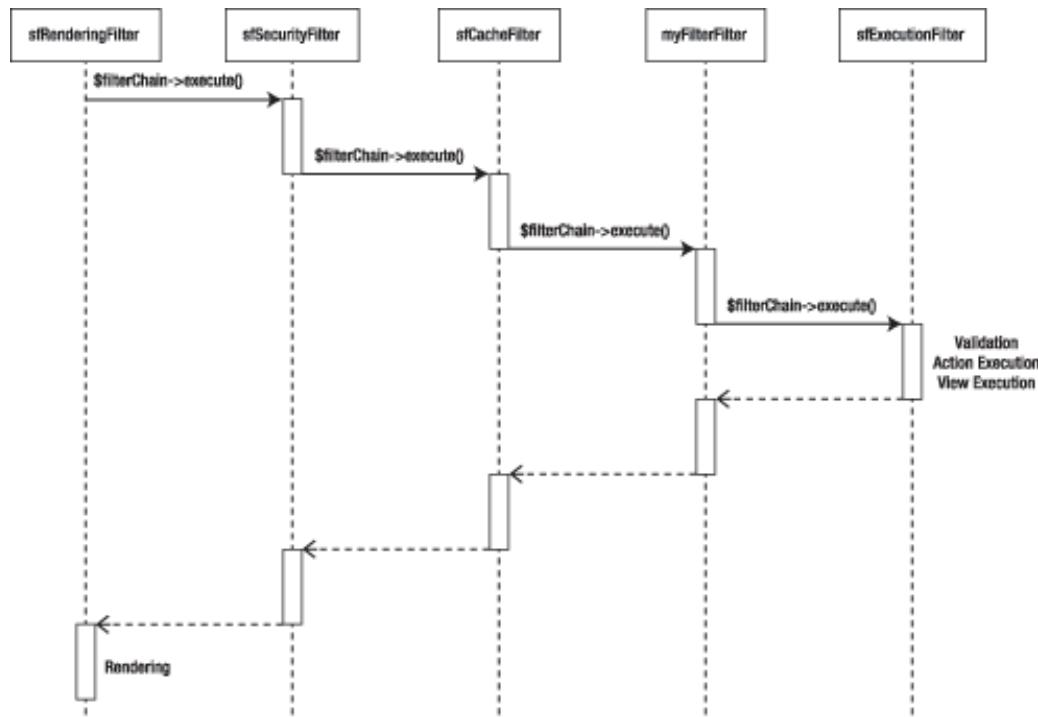
Filters

The security process can be understood as a filter by which all requests must pass before executing the action. According to some tests executed in the filter, the processing of the request is modified—for instance, by changing the action executed (`default/secure` instead of the requested action in the case of the security filter). Symfony extends this idea to filter classes. You can specify any number of filter classes to be executed before the action execution or before the response rendering, and do this for every request. You can see filters as a way to package some code, similar to `preExecute()` and `postExecute()`, but at a higher level (for a whole application instead of for a whole module).

The Filter Chain

Symfony actually sees the processing of a request as a chain of filters. When a request is received by the framework, the first filter (which is always the `sfRenderingFilter`) is executed. At some point, it calls the next filter in the chain, then the next, and so on. When the last filter (which is always `sfExecutionFilter`) is executed, the previous filter can finish, and so on back to the rendering filter. Figure 6-3 illustrates this idea with a sequence diagram, using an artificially small filter chain (the real one contains more filters).

Figure 6-3 - Sample filter chain



This process justifies the structure of the filter classes. They all extend the `sfFilter` class, and contain one `execute()` method, expecting a `$filterChain` object as parameter. Somewhere in this method, the filter passes to the next filter in the chain by calling `$filterChain->execute()`. See Listing 6-26 for an example. So basically, filters are divided into two parts:

- The code before the call to `$filterChain->execute()` executes before the action execution.
- The code after the call to `$filterChain->execute()` executes after the action execution and before the rendering.

Listing 6-26 - Filter Class Structure

```

Listing 6-44 class myFilter extends sfFilter
{
    public function execute ($filterChain)
    {
        // Code to execute before the action execution
        ...

        // Execute next filter in the chain
        $filterChain->execute();

        // Code to execute after the action execution, before the rendering
        ...
    }
}
  
```

The default filter chain is defined in an application configuration file called `filters.yml`, and is shown in Listing 6-27. This file lists the filters that are to be executed for every request.

Listing 6-27 - Default Filter Chain, in frontend/config/filters.yml

```

Listing 6-45 rendering: ~
security: ~
  
```

```
# Generally, you will want to insert your own filters here

cache: ~
execution: ~
```

These declarations have no parameter (the tilde character, `~`, means “null” in YAML), because they inherit the parameters defined in the symfony core. In the core, symfony defines `class` and `param` settings for each of these filters. For instance, Listing 6-28 shows the default parameters for the `rendering` filter.

Listing 6-28 - Default Parameters of the rendering Filter, in sfConfig::get('sf_symfony_lib_dir')/config/config/filters.yml

```
rendering:
  class: sfRenderingFilter    # Filter class
  param:                      # Filter parameters
  type: rendering
```

Listing 6-46

By leaving the empty value (`~`) in the application `filters.yml`, you tell symfony to apply the filter with the default settings defined in the core.

You can customize the filter chain in various ways:

- Disable some filters from the chain by adding an `enabled: false` parameter. For instance, to disable the `security` filter, write:

```
security:
  enabled: false
```

Listing 6-47

- Do not remove an entry from the `filters.yml` to disable a filter; symfony would throw an exception in this case.
- Add your own declarations somewhere in the chain (usually after the `security` filter) to add a custom filter (as discussed in the next section). Be aware that the `rendering` filter must be the first entry, and the `execution` filter must be the last entry of the filter chain.
- Override the default class and parameters of the default filters (notably to modify the security system and use your own security filter).

Building Your Own Filter

It is pretty simple to build a filter. Create a class definition similar to the one shown in Listing 6-26, and place it in one of the project’s `lib/` folders to take advantage of the autoloading feature.

As an action can forward or redirect to another action and consequently relaunch the full chain of filters, you might want to restrict the execution of your own filters to the first action call of the request. The `isFirstCall()` method of the `sfFilter` class returns a Boolean for this purpose. This call only makes sense before the action execution.

These concepts are clearer with an example. Listing 6-29 shows a filter used to auto-log users with a specific `MyWebSite` cookie, which is supposedly created by the login action. It is a rudimentary but working way to implement the “remember me” feature offered in login forms.

Listing 6-29 - Sample Filter Class File, Saved in apps/frontend/lib/rememberFilter.class.php

```
class rememberFilter extends sfFilter
{
```

Listing 6-48

```

public function execute($filterChain)
{
    // Execute this filter only once
    if ($this->isFirstCall())
    {
        // Filters don't have direct access to the request and user objects.
        // You will need to use the context object to get them
        $request = $this->getContext()->getRequest();
        $user    = $this->getContext()->getUser();

        if ($request->getCookie('MyWebSite'))
        {
            // sign in
            $user->setAuthenticated(true);
        }
    }

    // Execute next filter
    $filterChain->execute();
}
}

```

In some cases, instead of continuing the filter chain execution, you will need to forward to a specific action at the end of a filter. `sfFilter` doesn't have a `forward()` method, but `sfController` does, so you can simply do that by calling the following:

Listing 6-49

```
return $this->getContext()->getController()->forward('mymodule',
'myAction');
```



The `sfFilter` class has an `initialize()` method, executed when the filter object is created. You can override it in your custom filter if you need to deal with filter parameters (defined in `filters.yml`, as described next) in your own way.

Filter Activation and Parameters

Creating a filter file is not enough to activate it. You need to add your filter to the filter chain, and for that, you must declare the filter class in the `filters.yml`, located in the application or in the module `config/` directory, as shown in Listing 6-30.

Listing 6-30 - Sample Filter Activation File, Saved in `apps/frontend/config/filters.yml`

Listing 6-50

```

rendering: ~
security: ~

remember:                      # Filters need a unique name
    class: rememberFilter
    param:
        cookie_name: MyWebSite
        condition:   %APP_ENABLE_REMEMBER_ME%

cache:   ~
execution: ~

```

When activated, the filter is executed for each request. The filter configuration file can contain one or more parameter definitions under the `param` key. The filter class has the

ability to get the value of these parameters with the `getParameter()` method. Listing 6-31 demonstrates how to get a filter parameter value.

Listing 6-31 - Getting the Parameter Value, in apps/frontend/lib/rememberFilter.class.php

```
class rememberFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // ...

        if ($request->getCookie($this->getParameter('cookie_name')))
        {
            // ...
        }

        // ...
    }
}
```

Listing 6-51

The `condition` parameter is tested by the filter chain to see if the filter must be executed. So your filter declarations can rely on an application configuration, just like the one in Listing 6-30. The remember filter will be executed only if your application `app.yml` shows this:

```
all:
  enable_remember_me: true
```

Listing 6-52

Sample Filters

The filter feature is useful to repeat code for every action. For instance, if you use a distant analytics system, you probably need to put a code snippet calling a distant tracker script in every page. You could put this code in the global layout, but then it would be active for all of the application. Alternatively, you could place it in a filter, such as the one shown in Listing 6-32, and activate it on a per-module basis.

Listing 6-32 - Google Analytics Filter

```
class sfGoogleAnalyticsFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // Nothing to do before the action
        $filterChain->execute();

        // Decorate the response with the tracker code
        $googleCode =
<script src="http://www.google-analytics.com/urchin.js" type="text/
javascript">
</script>
<script type="text/javascript">
    _uacct="UA-' . $this->getParameter('google_id') . '"; urchinTracker();
</script>;
        $response = $this->getContext()->getResponse();
        $response->setContent(str_ireplace('</body>', 
$googleCode.'</body>', $response->getContent()));
    }
}
```

Listing 6-53

Be aware that this filter is not perfect, as it should not add the tracker on responses that are not HTML.

Another example would be a filter that switches the request to SSL if it is not already, to secure the communication, as shown in Listing 6-33.

Listing 6-33 - Secure Communication Filter

```
Listing 6-54
class sfSecureFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $context = $this->getContext();
        $request = $context->getRequest();

        if (!$request->isSecure())
        {
            $secure_url = str_replace('http', 'https', $request->getUri());

            return $context->getController()->redirect($secure_url);
            // We don't continue the filter chain
        }
        else
        {
            // The request is already secure, so we can continue
            $filterChain->execute();
        }
    }
}
```

Filters are used extensively in plug-ins, as they allow you to extend the features of an application globally. Refer to Chapter 17 to learn more about plug-ins.

Module Configuration

A few module behaviors rely on configuration. To modify them, you must create a `module.yml` file in the module's `config/` directory and define settings on a per-environment basis (or under the `all:` header for all environments). Listing 6-34 shows an example of a `module.yml` file for the `mymodule` module.

Listing 6-34 - Module Configuration, in `apps/frontend/modules/mymodule/config/module.yml`

```
Listing 6-55
all:                      # For all environments
    enabled:              true
    is_internal:          false
    view_class:           sfPHP
    partial_view_class:   sf
```

The `enabled` parameter allows you to disable all actions of a module. All actions are redirected to the `module_disabled_module/module_disabled_action` action (as defined in `settings.yml`).

The `is_internal` parameter allows you to restrict the execution of all actions of a module to internal calls. For example, this is useful for mail actions that you must be able to call from another action, to send an e-mail message, but not from the outside.

The `view_class` parameter defines the view class. It must inherit from `sfView`. Overriding this value allows you to use other view systems, with other templating engines, such as Smarty.

The `partial_view_class` parameter defines the view class used for partials of this module. It must inherit from `sfPartialView`.

Summary

In symfony, the controller layer is split into two parts: the front controller, which is the unique entry point to the application for a given environment, and the actions, which contain the page logic. An action has the ability to determine how its view will be executed, by returning one of the `sfView` constants. Inside an action, you can manipulate the different elements of the context, including the request object (`sfRequest`) and the current user session object (`sfUser`).

Combining the power of the session object, the action object, and the security configuration, symfony provides a complete security system, with access restriction and credentials. And if the `preExecute()` and `postExecute()` methods are made for reusability of code inside a module, the filters authorize the same reusability for all the applications by making controller code executed for every request.

Chapter 7

Inside The View Layer

The view is responsible for rendering the output correlated to a particular action. In symfony, the view consists of several parts, with each part designed to be easily modified by the person who usually works with it.

- Web designers generally work on the templates (the presentation of the current action data) and on the layout (containing the code common to all pages). These are written in HTML with small embedded chunks of PHP, which are mostly calls to helpers.
- For reusability, developers usually package template code fragments into partials or components. They use slots to affect more than one zone of the layout. Web designers can work on these template fragments as well.
- Developers focus on the YAML view configuration file (setting the properties of the response and other interface elements) and on the response object. When dealing with variables in the templates, the risks of cross-site scripting must not be ignored, and a good comprehension of output escaping techniques is required to safely record user data.

But whatever your role is, you will find useful tools to speed up the tedious job of presenting the results of the action. This chapter covers all of these tools.

Templating

Listing 7-1 shows a typical symfony template. It contains some HTML code and some basic PHP code, usually calls to variables defined in the action (via `$this->name = 'foo';`) and helpers.

Listing 7-1 - A Sample indexSuccess.php Template

Listing
7-1

```
<h1>Welcome</h1>
<p>Welcome back, <?php echo $name ?>!</p>
<h2>What would you like to do?</h2>
<ul>
  <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>
```

As explained in Chapter 4, the alternative PHP syntax is preferable for templates to make them readable for non-PHP developers. You should keep PHP code to a minimum 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. Keeping the logic inside the action also makes it easier to have several templates for a single action, without any code duplication.

Helpers

Helpers are PHP functions that return HTML code and can be used in templates. In Listing 7-1, the `link_to()` function is a helper. Sometimes, helpers are just time-savers, packaging code snippets frequently used in templates. For instance, you can easily imagine the function definition for this helper:

```
<?php echo image_tag('photo.jpg') ?>
=> 
```

Listing 7-2

It should look like Listing 7-2.

Listing 7-2 - Sample Helper Definition

```
function image_tag($source)
{
    return '';
}
```

Listing 7-3

As a matter of fact, the `image_tag()` function built into symfony is a little more complicated than that, as it accepts a second parameter to add other attributes to the `` tag. You can check its complete syntax and options in the online API documentation³⁶.

Most of the time, helpers carry intelligence and save you long and complex coding:

```
<?php echo auto_link_text('Please visit our website www.example.com') ?>
=> Please visit our website <a
href="http://www.example.com">www.example.com</a>
```

Listing 7-4

Helpers facilitate the process of writing templates and produce the best possible HTML code in terms of performance and accessibility. You can always use plain HTML, but helpers are usually faster to write.



You may wonder why the helpers are named according to the underscore syntax rather than the camelCase convention, used everywhere else in symfony. This is because helpers are functions, and all the core PHP functions use the underscore syntax convention.

Declaring Helpers

The symfony files containing helper definitions are not autoloaded (since they contain functions, not classes). Helpers are grouped by purpose. For instance, all the helper functions dealing with text are defined in a file called `TextHelper.php`, called the Text helper group. So if you need to use a helper in a template, you must load the related helper group earlier in the template by declaring it with the `use_helper()` function. Listing 7-3 shows a template using the `auto_link_text()` helper, which is part of the Text helper group.

Listing 7-3 - Declaring the Use of a Helper

```
// Use a specific helper group in this template
<?php use_helper('Text') ?>
...
<h1>Description</h1>
<p><?php echo auto_link_text($description) ?></p>
```

Listing 7-5

36. <http://www.symfony-project.org/api/1.4/>



If you need to declare more than one helper group, add more arguments to the `use_helper()` call. For instance, to load both the `Text` and the `Javascript` helper groups in a template, call `<?php use_helper('Text', 'Javascript') ?>`.

A few helpers are available by default in every template, without need for declaration. These are helpers of the following helper groups:

- **Helper:** Required for helper inclusion (the `use_helper()` function is, in fact, a helper itself)
- **Tag:** Basic tag helper, used by almost every helper
- **Url:** Links and URL management helpers
- **Asset:** Helpers populating the HTML `<head>` section, and providing easy links to external assets (images, JavaScript, and style sheet files)
- **Partial:** Helpers allowing for inclusion of template fragments
- **Cache:** Manipulation of cached code fragments

The list of the standard helpers, loaded by default for every template, is configurable in the `settings.yml` file. So if you know that you will not use the helpers of the `Cache` group, or that you will always use the ones of the `Text` group, modify the `standard_helpers` setting accordingly. This will speed up your application a bit. You cannot remove the first four helper groups in the preceding list (`Helper`, `Tag`, `Url`, and `Asset`), because they are compulsory for the templating engine to work properly. Consequently, they don't even appear in the list of standard helpers.



If you ever need to use a helper outside a template, you can still load a helper group from anywhere by calling `sfProjectConfiguration::getActive()->loadHelpers($helpers)`, where `$helpers` is a helper group name or an array of helper group names. For instance, if you want to use `auto_link_text()` in an action, you need to call `sfProjectConfiguration::getActive()->loadHelpers('Text')` first.

Frequently Used Helpers

You will learn about some helpers in detail in later chapters, in relation with the feature they are helping. Listing 7-4 gives a brief list of the default helpers that are used a lot, together with the HTML code they return.

Listing 7-4 - Common Default Helpers

```
Listing 7-6 // Helper group
<?php use_helper('HelperName') ?>
<?php use_helper('HelperName1', 'HelperName2', 'HelperName3') ?>

// Url group
<?php echo link_to('click me', 'mymodule/myaction') ?>
=> <a href="/route/to/myaction">click me</a> // Depends on the routing
settings

// Asset group
<?php echo image_tag('myimage', 'alt=foo size=200x100') ?>
=> 
<?php echo javascript_include_tag('myscript') ?>
=> <script language="JavaScript" type="text/javascript"
src="http://www.symfony-project.org/js/myscript.js"></script>
<?php echo stylesheet_tag('style') ?>
```

```
=> <link href="/stylesheets/style.css" media="screen"
rel="stylesheet" type="text/css" />
```

There are many other helpers in symfony, and it would take a full book to describe all of them. The best reference for helpers is the online [API documentation](http://www.symfony-project.org/api/1_4/), where all the helpers are well documented, with their syntax, options, and examples.

Adding Your Own Helpers

Symfony ships with a lot of helpers for various purposes, but if you don't find what you need in the API documentation, you will probably want to create a new helper. This is very easy to do.

Helper functions (regular PHP functions returning HTML code) should be saved in a file called `FooBarHelper.php`, where `FooBar` is the name of the helper group. Store the file in the `apps/frontend/lib/helper/` directory (or in any `helper/` directory created under one of the `lib/` folders of your project) so it can be found automatically by the `use_helper('FooBar')` helper for inclusion.



This system even allows you to override the existing symfony helpers. For instance, to redefine all the helpers of the `Text` helper group, just create a `TextHelper.php` file in your `apps/frontend/lib/helper/` directory. Whenever you call `use_helper('Text')`, symfony will use your helper group rather than its own. But be careful: as the original file is not even loaded, you must redefine all the functions of a helper group to override it; otherwise, some of the original helpers will not be available at all.

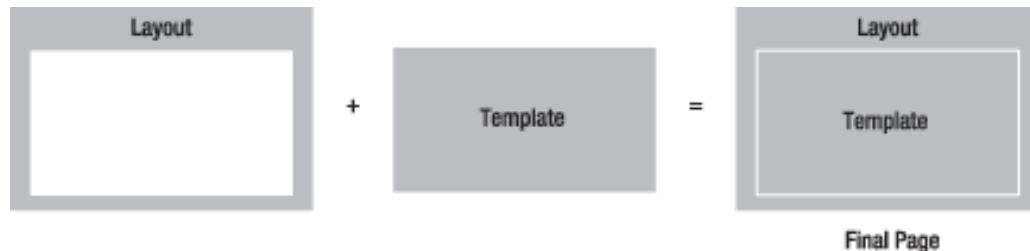
Page Layout

The template shown in Listing 7-1 is not a valid XHTML document. The `DOCTYPE` definition and the `<html>` and `<body>` tags are missing. That's because they are stored somewhere else in the application, in a file called `layout.php`, which contains the page layout. This file, also called the global template, stores the HTML code that is common to all pages of the application to avoid repeating it in every template. The content of the template is integrated into the layout, or, if you change the point of view, the layout "decorates" the template. This is an application of the decorator design pattern, illustrated in Figure 7-1.



For more information about the decorator and other design patterns, see *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley, ISBN: 0-32112-742-0).

Figure 7-1 - Decorating a template with a layout



Listing 7-5 shows the default page layout, located in the application `templates/` directory.

Listing 7-5 - Default Layout, in myproject/apps/frontend/templates/layout.php

Listing 7-7

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <?php include_javascripts() ?>
    <?php include_stylesheets() ?>
    <?php include_http_metas() ?>
    <?php include_metas() ?>
    <?php include_title() ?>
    <link rel="shortcut icon" href="/favicon.ico" />
  </head>
  <body>
    <?php echo $sf_content ?>
  </body>
</html>
```

The helpers called in the `<head>` section grab information from the response object and the view configuration. The `<body>` tag outputs the result of the template. With this layout, the default configuration, and the sample template in Listing 7-1, the processed view looks like Listing 7-6.

Listing 7-6 - The Layout, the View Configuration, and the Template Assembled

Listing 7-8

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<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" href="/css/main.css" />
    <link rel="shortcut icon" href="/favicon.ico">
  </head>
  <body>
    <h1>Welcome</h1>
    <p>Welcome back, <?php echo $name ?>!</p>
    <h2>What would you like to do?</h2>
    <ul>
      <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>
```

The global template can be entirely customized for each application. Add in any HTML code you need. This layout is often used to hold the site navigation, logo, and so on. You can even have more than one layout, and decide which layout should be used for each action. Don't worry about JavaScript and style sheet inclusion for now; the "View Configuration" section later in this chapter shows how to handle that.

Template Shortcuts

In templates, a few symfony variables are always available. These shortcuts give access to the most commonly needed information in templates, through the core symfony objects:

- `$sf_context`: The whole context object (*instance of sfContext*)
- `$sf_request`: The request object (*instance of sfRequest*)
- `$sf_params` : The parameters of the request object
- `$sf_user` : The current user session object (*instance of sfUser*)

The previous chapter detailed useful methods of the `sfRequest` and `sfUser` objects. You can actually call these methods in templates through the `$sf_request` and `$sf_user` variables. For instance, if the request includes a `total` parameter, its value is available in the template with the following:

```
// Long version
<?php echo $sf_request->getParameter('total') ?>

// Shorter version
<?php echo $sf_params->get('total') ?>

// Equivalent to the following action code
echo $request->getParameter('total')
```

*Listing
7-9*

Code Fragments

You may often need to include some HTML or PHP code in several pages. To avoid repeating that code, the PHP `include()` statement will suffice most of the time.

For instance, if many of the templates of your application need to use the same fragment of code, save it in a file called `myFragment.php` in the global template directory (`myproject/apps/frontend/templates/`) and include it in your templates as follows:

```
<?php include(sfConfig::get('sf_app_template_dir').'/myFragment.php') ?>
```

*Listing
7-10*

But this is not a very clean way to package a fragment, mostly because you can have different variable names between the fragment and the various templates including it. In addition, the symfony cache system (described in Chapter 12) has no way to detect an include, so the fragment cannot be cached independently from the template. Symfony provides three alternative types of intelligent code fragments to replace `includes`:

- 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 fragment is meant to replace a specific part of the layout, for which default content may already exist, you will use a slot.

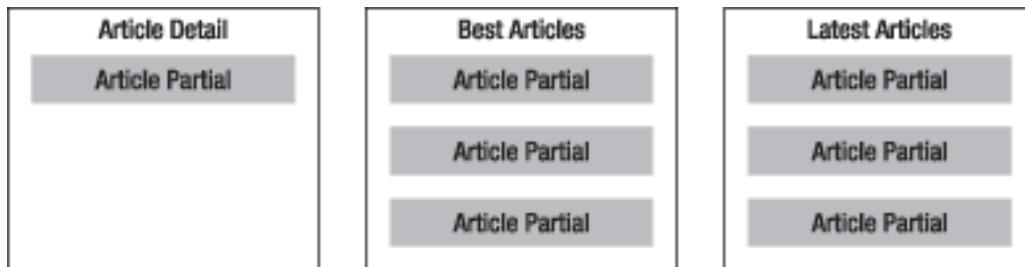
The inclusion of these fragments is achieved by helpers of the `Partial` group. These helpers are available from any symfony template, without initial declaration.

Partials

A partial is a reusable chunk of template code. For instance, in a publication application, the template code displaying an article is used in the article detail page, and also in the list of the

best articles and the list of latest articles. This code is a perfect candidate for a partial, as illustrated in Figure 7-2.

Figure 7-2 - Reusing partials in templates



Just like templates, partials are files located in the `templates/` directory, and they contain HTML code with embedded PHP. A partial file name always starts with an underscore (`_`), and that helps to distinguish partials from templates, since they are located in the same `templates/` folders.

A template can include partials whether it is in the same module, in another module, or in the global `templates/` directory. Include a partial by using the `include_partial()` helper, and specify the module and partial name as a parameter (but omit the leading underscore and the trailing `.php`), as described in Listing 7-7.

Listing 7-7 - Including a Partial in a Template of the mymodule Module

```

Listing 7-11
// Include the frontend/modules/mymodule/templates/_mypartial1.php partial
// As the template and the partial are in the same module,
// you can omit the module name
<?php include_partial('mypartial1') ?>

// Include the frontend/modules/foobar/templates/_mypartial2.php partial
// The module name is compulsory in that case
<?php include_partial('foobar/mypartial2') ?>

// Include the frontend/templates/_mypartial3.php partial
// It is considered as part of the 'global' module
<?php include_partial('global/mypartial3') ?>
  
```

Partials have access to the usual symfony helpers and template shortcuts. But since partials can be called from anywhere in the application, they do not have automatic access to the variables defined in the action calling the templates that includes them, unless passed explicitly as an argument. For instance, if you want a partial to have access to a `$total` variable, the action must hand it to the template, and then the template to the helper as a second argument of the `include_partial()` call, as shown in Listings 7-8, 7-9, and 7-10.

Listing 7-8 - The Action Defines a Variable, in mymodule/actions/actions.class.php

```

Listing 7-12
class mymoduleActions extends sfActions
{
    public function executeIndex()
    {
        $this->total = 100;
    }
}
  
```

Listing 7-9 - The Template Passes the Variable to the Partial, in mymodule/templates/indexSuccess.php

Listing 7-13

```
<p>Hello, world!</p>
<?php include_partial('mypartial', array('mytotal' => $total)) ?>
```

Listing 7-10 - The Partial Can Now Use the Variable, in mymodule/templates/_mypartial.php

```
<p>Total: <?php echo $mytotal ?></p>
```

*Listing
7-14*



All the helpers so far were called by `<?php echo functionName() ?>`. The partial helper, however, is simply called by `<?php include_partial() ?>`, without echo, to make it behave similar to the regular PHP `include()` statement. If you ever need a function that returns the content of a partial without actually displaying it, use `get_partial()` instead. All the `include_` helpers described in this chapter have a `get_` counterpart that can be called together with an `echo` statement.

TIP Instead of resulting in a template, an action can return a partial or a component. The `renderPartial()` and `renderComponent()` methods of the action class promote reusability of code. Besides, they take advantage of the caching abilities of the partials (see Chapter 12). The variables defined in the action will be automatically passed to the partial/component, unless you define an associative array of variables as a second parameter of the method.

```
public function executeFoo()
{
    // do things
    $this->foo = 1234;
    $this->bar = 4567;

    return $this->renderPartial('mymodule/mypartial');
}
```

*Listing
7-15*

In this example, the partial will have access to `$foo` and `$bar`. If the action ends with the following line:

```
return $this->renderPartial('mymodule/mypartial', array('foo' =>
$this->foo));
```

*Listing
7-16*

Then the partial will only have access to `$foo`.

Components

In Chapter 2, the first sample script was split into two parts to separate the logic from the presentation. Just like the MVC pattern applies to actions and templates, you may need to split a partial into a logic part and a presentation part. In such a case, you should use a component.

A component is like an action, except it's much faster. The logic of a component is kept in a class inheriting from `sfComponents`, located in an `actions/components.class.php` file. Its presentation is kept in a partial. Methods of the `sfComponents` class start with the word `execute`, just like actions, and they can pass variables to their presentation counterpart in the same way that actions can pass variables. Partial that serve as presentation for components are named by the component (without the leading `execute`, but with an underscore instead). Table 7-1 compares the naming conventions for actions and components.

Table 7-1 - Action and Component Naming Conventions

Convention	Actions	Components
Logic file	<code>actions.class.php</code>	<code>components.class.php</code>

Convention	Actions	Components
Logic class extends	<code>sfActions</code>	<code>sfComponents</code>
Method naming	<code>executeMyAction()</code>	<code>executeMyComponent()</code>
Presentation file naming	<code>myActionSuccess.php</code>	<code>_myComponent.php</code>

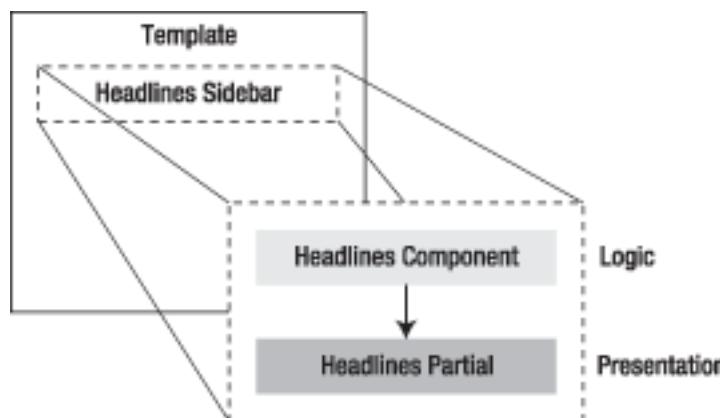


Just as you can separate actions files, the `sfComponents` class has an `sfComponent` counterpart that allows for single component files with the same type of syntax.

For instance, suppose you have a sidebar displaying the latest news headlines for a given subject, depending on the user's profile, which is reused in several pages. The queries necessary to get the news headlines are too complex to appear in a simple partial, so they need to be moved to an action-like file—a component. Figure 7-3 illustrates this example.

For this example, shown in Listings 7-11 and 7-12, the component will be kept in its own module (called `news`), but you can mix components and actions in a single module if it makes sense from a functional point of view.

Figure 7-3 - Using components in templates



Listing 7-11 - The Components Class, in `modules/news/actions/components.class.php`

Listing 7-11 <?php

```

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

        // Doctrine
        $query = Doctrine::getTable('News')
            ->createQuery()
            ->orderBy('published_at DESC')
            ->limit(5);

        $this->news = $query->execute();
    }
}
  
```

Listing 7-12 - The Partial, in modules/news/templates/_headlines.php

```
<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>
```

Listing
7-18

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

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

Listing
7-19

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. See Listing 7-13 for an example.

Listing 7-13 - Passing Parameters to a Component and Its Template

```
// 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'
```

Listing
7-20

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 component doesn't handle security or validation, cannot be called from the Internet (only from the application itself), and doesn't have various return possibilities. That's why a component is faster to execute than an action.

Slots

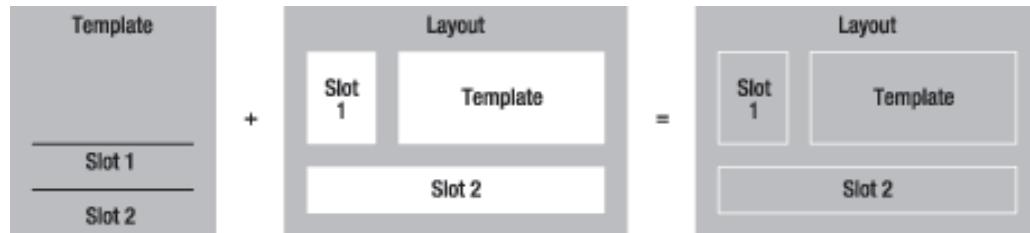
Partials and components are great for reusability. But in many cases, code fragments are required to fill a layout with more than one dynamic zone. For instance, suppose that you want to add some custom tags in the `<head>` section of the layout, depending on the content of the action. Or, suppose that the layout has one major dynamic zone, which is filled by the result of the action, plus a lot of other smaller ones, which have a default content defined in the layout but can be overridden at the template level.

For these situations, the solution is a slot. Basically, a slot is a placeholder that you can put in any of the view elements (in the layout, a template, or a partial). Filling this placeholder is just like setting a variable. The filling code is stored globally in the response, so you can define it anywhere (in the layout, a template, or a partial). Just make sure to define a slot before including it, and remember that the layout is executed after the template (this is the

decoration process), and the partials are executed when they are called in a template. Does it sound too abstract? Let's see an example.

Imagine a layout with one zone for the template and two slots: one for the sidebar and the other for the footer. The slot values are defined in the templates. During the decoration process, the layout code wraps the template code, and the slots are filled with the previously defined values, as illustrated in Figure 7-4. The sidebar and the footer can then be contextual to the main action. This is like having a layout with more than one "hole".

Figure 7-4 - Layout slots defined in a template



Seeing some code will clarify things further. To include a slot, use the `include_slot()` helper. The `has_slot()` helper returns `true` if the slot has been defined before, providing a fallback mechanism as a bonus. For instance, define a placeholder for a '`'sidebar'` slot in the layout and its default content as shown in Listing 7-14.

Listing 7-14 - Including a 'sidebar' Slot in the Layout

```

Listing 7-21
<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>
  
```

As it's quite common to display some default content if a slot is not defined, the `include_slot` helper returns a Boolean indicating if the slot has been defined. Listing 7-15 shows how to take this return value into account to simplify the code.

Listing 7-15 - Including a 'sidebar' Slot in the Layout

```

Listing 7-22
<div id="sidebar">
<?php if (!include_slot('sidebar')): ?>
  <!-- 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>
  
```

Each template has the ability to define the contents of a slot (actually, even partials can do it). As slots are meant to hold HTML code, symfony offers a convenient way to define them: just write the slot code between a call to the `slot()` and `end_slot()` helpers, as in Listing 7-16.

Listing 7-16 - Overriding the 'sidebar' Slot Content in a Template

```

Listing 7-23
// ...
<?php slot('sidebar') ?>
  
```

```
<!-- custom sidebar code for the current template-->
<h1>User details</h1>
<p>name: <?php echo $user->getName() ?></p>
<p>email: <?php echo $user->getEmail() ?></p>
<?php end_slot() ?>
```

The code between the slot helpers is executed in the context of the template, so it has access to all the variables that were defined in the action. Symfony will automatically put the result of this code in the response object. It will not be displayed in the template, but made available for future `include_slot()` calls, like the one in Listing 7-14.

Slots are very useful to define zones meant to display contextual content. They can also be used to add HTML code to the layout for certain actions only. For instance, a template displaying the list of the latest news might want to add a link to an RSS feed in the `<head>` part of the layout. This is achieved simply by adding a 'feed' slot in the layout and overriding it in the template of the list.

If the content of the slot is very short, as this is the case when defining a `title` slot for example, you can simply pass the content as a second argument of the `slot()` method as shown in Listing 7-17.

Listing 7-17 - Using `slot()` to define a short Value

```
<?php slot('title', 'The title value') ?>
```

*Listing
7-24*

Where to find template fragments

People working on templates are usually web designers, who may not know symfony very well and may have difficulties finding template fragments, since they can be scattered all over the application. These few guidelines will make them more comfortable with the symfony templating system.

First of all, although a symfony project contains many directories, all the layouts, templates, and template fragments files reside in directories named `templates/`. So as far as a web designer is concerned, a project structure can be reduced to something like this:

```
myproject/
  apps/
    application1/
      templates/      # Layouts for application 1
      modules/
        module1/
          templates/  # Templates and partials for module 1
        module2/
          templates/  # Templates and partials for module 2
        module3/
          templates/  # Templates and partials for module 3
```

*Listing
7-25*

All other directories can be ignored.

When meeting an `include_partial()`, web designers just need to understand that only the first argument is important. This argument's pattern is `module_name/partial_name`, and that means that the presentation code is to be found in `modules/module_name/templates/_partial_name.php`.

For the `include_component()` helper, module name and partial name are the first two arguments. As for the rest, a general idea about what helpers are and which helpers are the most common in templates should be enough to start designing templates for symfony applications.

View Configuration

In symfony, a view consists of two distinct parts:

- The HTML presentation of the action result (stored in the template, in the layout, and in the template fragments)
- All the rest, including the following:
 - 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 style sheet files.
 - Layout: Some actions require a custom layout (pop-ups, ads, and so on) or no layout at all (such as Ajax actions).

In the view, all that is not HTML is called view configuration, and symfony provides two ways to manipulate it. The usual way is through the `view.yml` configuration file. It can be used whenever the values don't depend on the context or on database queries. When you need to set dynamic values, the alternative method is to set the view configuration via the `sfResponse` object attributes directly in the action.



If you ever set a view configuration parameter both via the `sfResponse` object and via the `view.yml` file, the `sfResponse` definition takes precedence.

The `view.yml` File

Each module can have one `view.yml` file defining the settings of its views. This allows you to define view settings for a whole module and per view in a single file. The first-level keys of the `view.yml` file are the module view names. Listing 7-18 shows an example of view configuration.

Listing 7-18 - Sample Module-Level `view.yml`

```
Listing 7-26
editSuccess:
  metas:
    title: Edit your profile

editError:
  metas:
    title: Error in the profile edition

all:
  stylesheets: [my_style]
  metas:
    title: My website
```



Be aware that the main keys in the `view.yml` file are view names, not action names. As a reminder, a view name is composed of an action name and an action termination. For instance, if the `edit` action returns `sfView::SUCCESS` (or returns nothing at all, since it is the default action termination), then the view name is `editSuccess`.

The default settings for the module are defined under the `all:` key in the module `view.yml`. The default settings for all the application views are defined in the application `view.yml`. Once again, you recognize the configuration cascade principle:

- In `apps/frontend/modules/mymodule/config/view.yml`, the per-view definitions apply only to one view and override the module-level definitions.
- In `apps/frontend/modules/mymodule/config/view.yml`, the `all:` definitions apply to all the actions of the module and override the application-level definitions.
- In `apps/frontend/config/view.yml`, the `default:` definitions apply to all modules and all actions of the application.



Module-level `view.yml` files don't exist by default. The first time you need to adjust a view configuration parameter for a module, you will have to create an empty `view.yml` in its `config` directory.

After seeing the default template in Listing 7-5 and an example of a final response in Listing 7-6, you may wonder where the header values come from. As a matter of fact, they are the default view settings, defined in the application `view.yml` and shown in Listing 7-19.

Listing 7-19 - Default Application-Level View Configuration, in `apps/frontend/config/view.yml`

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

  metas:
    #title: symfony project
    #description: symfony project
    #keywords: symfony, project
    #language: en
    #robots: index, follow

  stylesheets: [main]

  javascripts: []

  has_layout: true
  layout: layout
```

*Listing
7-27*

Each of these settings will be described in detail in the “View Configuration Settings” section.

The Response Object

Although part of the view layer, the response object is often modified by the action. Actions can access the symfony response object, called `sfResponse`, via the `getResponse()` method. Listing 7-20 lists some of the `sfResponse` methods often used from within an action.

Listing 7-20 - Actions Have Access to the `sfResponse` Object Methods

```
class mymoduleActions extends sfActions
{
  public function executeIndex()
  {
    $response = $this->getResponse();

    // HTTP headers
    $response->setContentType('text/xml');
    $response->setHttpHeader('Content-Language', 'en');
    $response->setStatusCode(403);
    $response->addVaryHttpHeader('Accept-Language');
```

*Listing
7-28*

```
$response->addCacheControlHTTPHeader('no-cache');

// Cookies
$response->setCookie($name, $content, $expire, $path, $domain);

// Metas and page headers
$response->addMeta('robots', 'NONE');
$response->addMeta('keywords', 'foo bar');
$response->setTitle('My FooBar Page');
$response->addStyleSheet('custom_style');
$response->addJavaScript('custom_behavior');

}

}
```

In addition to the setter methods shown here, the `sfResponse` class has getters that return the current value of the response attributes.

The header setters are very powerful in symfony. Headers are sent as late as possible (in the `sfRenderingFilter`), so you can alter them as much as you want and as late as you want. They also provide very useful shortcuts. For instance, if you don't specify a charset when you call `setContent-Type()`, symfony automatically adds the default charset defined in the `settings.yml` file.

Listing 7-29

```
$response->setContent-Type('text/xml');
echo $response->getContent-Type();
=> 'text/xml; charset=utf-8'
```

The status code of responses in symfony is compliant with the HTTP specification. Exceptions return a status 500, pages not found return a status 404, normal pages return a status 200, pages not modified can be reduced to a simple header with status code 304 (see Chapter 12 for details), and so on. But you can override these defaults by setting your own status code in the action with the `setStatus-Code()` response method. You can specify a custom code and a custom message, or simply a custom code—in which case, symfony will add the most common message for this code.

Listing 7-30

```
$response->setStatus-Code(404, 'This page does not exist');
```



Before sending the headers, symfony normalizes their names. So you don't need to bother about writing `content-language` instead of `Content-Language` in a call to `setHttpHeader()`, as symfony will understand the former and automatically transform it to the latter.

View Configuration Settings

You may have noticed that there are two kinds of view configuration settings:

- The ones that have a unique value (the value is a string in the `view.yml` file and the response uses a `set` method for those)
- The ones with multiple values (for which `view.yml` uses arrays and the response uses an `add` method)

Keep in mind that the configuration cascade erases the unique value settings but piles up the multiple values settings. This will become more apparent as you progress through this chapter.

Meta Tag Configuration

The information written in the `<meta>` tags in the response is not displayed in a browser but is useful for robots and search engines. It also controls the cache settings of every page. Define these tags under the `http_metas:` and `metas:` keys in `view.yml`, as in Listing 7-21, or with the `addHttpMeta()` and `addMeta()` response methods in the action, as in Listing 7-22.

Listing 7-21 - Meta Definition As Key: Value Pairs in view.yml

```
http_metas:
    cache-control: public

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

*Listing
7-31*

Listing 7-22 - Meta Definition As Response Settings in the Action

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

*Listing
7-32*

Adding an existing key will replace its current content by default. For HTTP meta tags, you can add a third parameter and set it to `false` to have the `addHttpMeta()` method (as well as the `setHttpHeader()`) append the value to the existing one, rather than replacing it.

```
$this->getResponse()->addHttpMeta('accept-language', 'en');
$this->getResponse()->addHttpMeta('accept-language', 'fr', false);
echo $this->getResponse()->getHttpHeader('accept-language');
=> 'en, fr'
```

*Listing
7-33*

In order to have these meta tags appear in the final document, the `include_http_metas()` and `include_metas()` helpers must be called in the `<head>` section (this is the case in the default layout; see Listing 7-5). Symfony automatically aggregates the settings from all the `view.yml` files (including the default one shown in Listing 7-18) and the response attribute to output proper `<meta>` tags. The example in Listing 7-21 ends up as shown in Listing 7-23.

Listing 7-23 - Meta Tags Output in the Final Page

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta http-equiv="cache-control" content="public" />
<meta name="robots" content="index, follow" />
<meta name="description" content="Finance in France" />
<meta name="keywords" content="finance, France" />
```

*Listing
7-34*

As a bonus, the HTTP header of the response is also impacted by the `http-metas:` definition, even if you don't have any `include_http_metas()` helpers in the layout, or if you have no layout at all. For instance, if you need to send a page as plain text, define the following `view.yml`:

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

has_layout: false
```

*Listing
7-35*

Title Configuration

The page title is a key part to search engine indexing. It is also very useful with modern browsers that provide tabbed browsing. In HTML, the title is both a tag and meta information of the page, so the `view.yml` file sees the `title:` key as a child of the `metas:` key. Listing 7-24 shows the title definition in `view.yml`, and Listing 7-25 shows the definition in the action.

Listing 7-24 - Title Definition in view.yml

```
Listing 7-36 indexSuccess:
  metas:
    title: Three little piggies
```

Listing 7-25 - Title Definition in the Action—Allows for Dynamic Titles

```
Listing 7-37 $this->getResponse()->setTitle(sprintf('%d little piggies', $number));
```

In the `<head>` section of the final document, the title definition sets the `<meta name="title">` tag if the `include_metas()` helper is present, and the `<title>` tag if the `include_title()` helper is present. If both are included (as in the default layout of Listing 7-5), the title appears twice in the document source (see Listing 7-6), which is harmless.



Another way to handle the title definition is to use slots, as discussed above. This method allows to keep a better separation of concern between controllers and templates : the title belongs to the view, not the controller.

File Inclusion Configuration

Adding a specific style sheet or JavaScript file to a view is easy, as Listing 7-26 demonstrates.

Listing 7-26 - Asset File Inclusion

```
Listing 7-38 // In the view.yml
indexSuccess:
  stylesheets: [mystyle1, mystyle2]
  javascripts: [myscript]

Listing 7-39 // In the action
$this->getResponse()->addStylesheet('mystyle1');
$this->getResponse()->addStylesheet('mystyle2');
$this->getResponse()->addJavascript('myscript');

// In the Template
<?php use_stylesheet('mystyle1') ?>
<?php use_stylesheet('mystyle2') ?>
<?php use_javascript('myscript') ?>
```

In each case, the argument is a file name. If the file has a logical extension (`.css` for a style sheet and `.js` for a JavaScript file), you can omit it. If the file has a logical location (`/css/` for a style sheet and `/js/` for a JavaScript file), you can omit it as well. Symfony is smart enough to figure out the correct extension or location.

Like the meta and title definitions, the file inclusion definitions require the usage of the `include_javascripts()` and `include_stylesheets()` helpers in the template or layout to be included. This means that the previous settings will output the HTML code of Listing 7-27.

Listing 7-27 - File Inclusion Result

```
<head>
...
<link rel="stylesheet" type="text/css" media="screen" href="/css/
mystyle1.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/css/
mystyle2.css" />
<script language="javascript" type="text/javascript"
src="http://www.symfony-project.org/js/myscript.js">
</script>
</head>
```

Listing
7-40

Remember that the configuration cascade principle applies, so any file inclusion defined in the application `view.yml` makes it appear in every page of the application. Listings 7-28, 7-29, and 7-30 demonstrate this principle.

Listing 7-28 - Sample Application view.yml

```
default:
  stylesheets: [main]
```

Listing
7-41*Listing 7-29 - Sample Module view.yml*

```
indexSuccess:
  stylesheets: [special]

all:
  stylesheets: [additional]
```

Listing
7-42*Listing 7-30 - Resulting indexSuccess View*

```
<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" />
```

Listing
7-43

If you need to remove a file defined at a higher level, just add a minus sign (-) in front of the file name in the lower-level definition, as shown in Listing 7-31.

Listing 7-31 - Sample Module view.yml That Removes a File Defined at the Application Level

```
indexSuccess:
  stylesheets: [-main, special]

all:
  stylesheets: [additional]
```

Listing
7-44

To remove all style sheets or JavaScript files, use -* as a file name, as shown in Listing 7-32.

Listing 7-32 - Sample Module view.yml That Removes all Files Defined at the Application Level

```
indexSuccess:
  stylesheets: [-*]
  javascripts: [-*]
```

Listing
7-45

You can be more accurate and define an additional parameter to force the position where to include the file (first or last position), as shown in Listing 7-33. This works for both style sheets and JavaScript files.

Listing 7-33 - Defining the Position of the Included Asset

```
Listing // In the view.yml
7-46 indexSuccess:
    stylesheets: [special: { position: first }]

Listing // In the action
7-47 $this->getResponse()->addStylesheet('special', 'first');

// In the template
```

You can also decide to bypass the transformation of the asset file name, so that the resulting `<link>` or `<script>` tags refer to the exact location specified, as show in Listing 7-34.

Listing 7-34 - Style Sheet Inclusion with Raw Name

```
Listing // In the view.yml
7-48 indexSuccess:
    stylesheets: [main, paper: { raw_name: true }]

Listing // In the Action
7-49 $this->getResponse()->addStylesheet('main', '', array('raw_name' => true));

// In the template
true) ?>

// Resulting View
<link rel="stylesheet" type="text/css" href="main" />
```

To specify media for a style sheet inclusion, you can change the default style sheet tag options, as shown in Listing 7-35.

Listing 7-35 - Style Sheet Inclusion with Media

```
Listing // In the view.yml
7-50 indexSuccess:
    stylesheets: [main, paper: { media: print }]

Listing // In the Action
7-51 $this->getResponse()->addStylesheet('paper', '', array('media' =>
'print'));

// In the template
'print') ?>

// Resulting View
<link rel="stylesheet" type="text/css" media="print" href="/css/paper.css"
/>
```

Note about assets inclusions using the view.yml file

The best practice is to define the defaults stylesheets and javascripts file in the project view.yml file, and include specifics stylesheets or javascripts files in your templates using the dedicated helper. This way, you do not have to remove or replace already included assets, which can be a painfull in some cases.

Layout Configuration

According to the graphical charter of your website, you may have several layouts. Classic websites have at least two: the default layout and the pop-up layout.

You have already seen that the default layout is `myproject/apps/frontend/templates/layout.php`. Additional layouts must be added in the same global `templates/` directory. If you want a view to use a `frontend/templates/my_layout.php` file, use the syntax shown in Listing 7-36.

Listing 7-36 - Layout Definition

```
// In view.yml
indexSuccess:
    layout: my_layout

// In the action
$this->setLayout('my_layout');

// In the template
<?php decorate_with('my_layout') ?>
```

*Listing
7-52*

*Listing
7-53*

Some views don't need any layout at all (for instance, plain text pages or RSS feeds). In that case, set `has_layout` to `false`, as shown in Listing 7-37.

Listing 7-37 - Layout Removal

```
// In `view.yml'
indexSuccess:
    has_layout: false

// In the Action
$this->setLayout(false);

// In the template
<?php decorate_with(false) ?>
```

*Listing
7-54*

*Listing
7-55*



Ajax actions views have no layout by default.

Output Escaping

When you insert dynamic data in a template, you must be sure about the data integrity. For instance, if data comes from forms filled in by anonymous users, there is a risk that it may include malicious scripts intended to launch cross-site scripting (XSS) attacks. You must be able to escape the output data, so that any HTML tag it contains becomes harmless.

As an example, suppose that a user fills an input field with the following value:

Listing 7-56 <script>alert(document.cookie)</script>

If you echo this value without caution, the JavaScript will execute on every browser and allow for much more dangerous attacks than just displaying an alert. This is why you must escape the value before displaying it, so that it becomes something like this:

Listing 7-57 <script&gtalert(document.cookie)</script&gt

You could escape your output manually by enclosing every unsure value in a call to `htmlspecialchars()`, but that approach would be very repetitive and error-prone. Instead, Symfony provides a special system, called output escaping, which automatically escapes every variable output in a template. It is activated by default in the application `settings.yml`.

Activating Output Escaping

Output escaping is configured globally for an application in the `settings.yml` file. Two parameters control the way that output escaping works: the strategy determines how the variables are made available to the view, and the method is the default escaping function applied to the data.

Basically, all you need to do to activate output escaping is to set the `escaping_strategy` parameter to `true` (which is the default), as shown in Listing 7-38.

Listing 7-38 - Activating Output Escaping, in frontend/config/settings.yml

Listing 7-58 all:
 .settings:
 escaping_strategy: true
 escaping_method: ESC_SPECIALCHARS

This will add `htmlspecialchars()` to all variable output by default. For instance, suppose that you define a `test` variable in an action as follows:

Listing 7-59 \$this->test = '<script>alert(document.cookie)</script>';

With output escaping turned on, echoing this variable in the template will output the escaped data:

Listing 7-60 echo \$test;
=> <script&gtalert(document.cookie)</script&gt;

In addition, every template has access to an `$sf_data` variable, which is a container object referencing all the escaped variables. So you can also output the test variable with the following:

Listing 7-61 echo \$sf_data->get('test');
=> <script&gtalert(document.cookie)</script&gt;



The `$sf_data` object implements the Array interface, so instead of using the `$sf_data->get('myvariable')`, you can retrieve escaped values by calling `$sf_data['myvariable']`. But it is not a real array, so functions like `print_r()` will not work as expected.

`$sf_data` also gives you access to the unescaped, or raw, data. This is useful when a variable stores HTML code meant to be interpreted by the browser, provided that you trust this variable. Call the `getRaw()` method when you need to output the raw data.

```
echo $sf_data->getRaw('test');
=> <script>alert(document.cookie)</script>
```

*Listing
7-62*

You will have to access raw data each time you need variables containing HTML to be really interpreted as HTML.

When `escaping_strategy` is `false`, `$sf_data` is still available, but it always returns raw data.

Escaping Helpers

Escaping helpers are functions returning an escaped version of their input. They can be provided as a default `escaping_method` in the `settings.yml` file or to specify an escaping method for a specific value in the view. The following escaping helpers are available:

- `ESC_RAW`: Doesn't escape the value.
- `ESC_SPECIALCHARS`: Applies the PHP function `htmlspecialchars()` to the input.
- `ESC_ENTITIES`: Applies the PHP function `htmlentities()` to the input with `ENT_QUOTES` as the quote style.
- `ESC_JS`: Escapes a value to be put into a JavaScript string that is going to be used as HTML. This is useful for escaping things where HTML is going to be dynamically changed using JavaScript.
- `ESC_JS_NO_ENTITIES`: Escapes a value to be put into a JavaScript string but does not add entities. This is useful if the value is going to be displayed using a dialog box (for example, for a `myString` variable used in `javascript:alert(myString);`).

Escaping Arrays and Objects

Output escaping not only works for strings, but also for arrays and objects. Any values that are objects or arrays will pass on their escaped state to their children. Assuming your strategy is set to `true`, Listing 7-39 demonstrates the escaping cascade.

Listing 7-39 - Escaping Also Works for Arrays and Objects

```
// Class definition
class myClass
{
    public function testSpecialChars($value = '')
    {
        return '<' . $value . '>';
    }
}

// In the action
$this->test_array = array('&', '<', '>');
$this->test_array_of_arrays = array(array('&'));
$this->test_object = new myClass();

// In the template
<?php foreach($test_array as $value): ?>
    <?php echo $value ?>
<?php endforeach; ?>
=> &amp; &lt; &gt;
<?php echo $test_array_of_arrays[0][0] ?>
=> &amp;
<?php echo $test_object->testSpecialChars('&') ?>
=> &lt;&amp;&gt;
```

*Listing
7-63*

As a matter of fact, the variables in the template are not of the type you might expect. The output escaping system “decorates” them and transforms them into special objects:

Listing 7-64

```
<?php echo get_class($test_array) ?>
=> sfOutputEscaperArrayDecorator
<?php echo get_class($test_object) ?>
=> sfOutputEscaperObjectDecorator
```

This explains why some usual PHP functions (like `array_shift()`, `print_r()`, and so on) don’t work on escaped arrays anymore. But they can still be accessed using `[]`, be traversed using `foreach`, and they give back the right result with `count()`. And in templates, the data should be read-only anyway, so most access will be through the methods that do work.

You still have a way to retrieve the raw data through the `$sf_data` object. In addition, methods of escaped objects are altered to accept an additional parameter: an escaping method. So you can choose an alternative escaping method each time you display a variable in a template, or opt for the `ESC_RAW` helper to deactivate escaping. See Listing 7-40 for an example.

Listing 7-40 - Methods of Escaped Objects Accept an Additional Parameter

Listing 7-65

```
<?php echo $test_object->testSpecialChars('&') ?>
=> &lt;&gt;;
// The three following lines return the same value
<?php echo $test_object->testSpecialChars('&', ESC_RAW) ?>
<?php echo $sf_data->getRaw('test_object')->testSpecialChars('&') ?>
<?php echo $sf_data->get('test_object', ESC_RAW)->testSpecialChars('&') ?>
=> <&>
```

If you deal with objects in your templates, you will use the additional parameter trick a lot, since it is the fastest way to get raw data on a method call.



The usual symfony variables are also escaped when you turn on output escaping. So be aware that `$sf_user`, `$sf_request`, `$sf_param`, and `$sf_context` still work, but their methods return escaped data, unless you add `ESC_RAW` as a final argument to their method calls.



Even if XSS is one of the most common exploit of websites, this is not the only one. CSRF is also very popular and symfony provides automatic forms protection. You will discover how this security protection works in the chapter 10.

Summary

All kinds of tools are available to manipulate the presentation layer. The templates are built in seconds, thanks to helpers. The layouts, partials, and components bring both modularity and reusability. The view configuration takes advantage of the speed of YAML to handle (mostly) page headers. The configuration cascade exempts you from defining every setting for each view. For every modification of the presentation that depends on dynamic data, the action has access to the `sfResponse` object. And the view is secure from XSS attacks, thanks to the output escaping system.

Chapter 8

Inside The Model Layer (Doctrine)

Much of the discussion so far has been devoted to building pages, and processing requests and responses. But the business logic of a web application relies mostly on its data model. Symfony's default model component is based on an object/relational mapping layer. Symfony comes bundles with the two most popular PHP ORMs: Propel³⁷ and Doctrine³⁸. In a symfony application, you access data stored in a database and modify it through objects; you never address the database explicitly. This maintains a high level of abstraction and portability.

This chapter explains how to create an object data model, and the way to access and modify the data in Doctrine. It also demonstrates the integration of Doctrine in Symfony.



If you want to use Propel instead of Doctrine, read Appendix A instead as it contains the exact same information but for Propel.

Why Use an ORM and an Abstraction Layer?

Databases are relational. PHP 5 and symfony are object-oriented. In order to most effectively access the database in an object-oriented context, an interface translating the object logic to the relational logic is required. As explained in Chapter 1, this interface is called an object-relational mapping (ORM), and it is made up of objects that give access to data and keep business rules within themselves.

The main benefit of an ORM is reusability, allowing the methods of a data object to be called from various parts of the application, even from different applications. The ORM layer also encapsulates the data logic—for instance, the calculation of a forum user rating based on how many contributions were made and how popular these contributions are. When a page needs to display such a user rating, it simply calls a method of the data model, without worrying about the details of the calculation. If the calculation changes afterwards, you will just need to modify the rating method in the model, leaving the rest of the application unchanged.

Using objects instead of records, and classes instead of tables, has another benefit: They allow you to add new accessors to your objects that don't necessarily match a column in a table. For instance, if you have a table called `client` with two fields named `first_name` and `last_name`, 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, as in Listing 8-1. From the application point of view, there is no difference between the `FirstName`, `LastName`, and `Name` attributes of the `Client` class. Only the class itself can determine which attributes correspond to a database column.

Listing 8-1 - Accessors Mask the Actual Table Structure in a Model Class

37. <http://www.propelorm.org/>

38. <http://www.doctrine-project.org/>

Listing 8-1

```
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. Suppose you have a `ShoppingCart` class in which you keep `Items` (which are objects). To get the full amount of the shopping cart for the checkout, write a custom method to encapsulate the actual calculation, as shown in Listing 8-2.

Listing 8-2 - Accessors Mask the Data Logic

Listing 8-2

```
public function getTotal()
{
    $total = 0;
    foreach ($this->getItems() as $item)
    {
        $total += $item->getPrice() * $item->getQuantity();
    }

    return $total;
}
```

There is another important point to consider when building data-access procedures: Database vendors use different SQL syntax variants. Switching to another database management system (DBMS) forces you to rewrite part of the SQL queries that were designed for the previous one. If you build your queries using a database-independent syntax, and leave the actual SQL translation to a third-party component, you can switch database systems without pain. This is the goal of the database abstraction layer. It forces you to use a specific syntax for queries, and does the dirty job of conforming to the DBMS particulars and optimizing the SQL code.

The main benefit of an abstraction layer is portability, because it makes switching to another database possible, even in the middle of a project. Suppose that you need 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 decide. Just change one line in a configuration file, and it works.

Symfony uses Propel or Doctrine as the ORM, and they use PHP Data Objects for database abstraction. These two third-party components, both developed by the Propel and Doctrine teams, are seamlessly integrated into symfony, and you can consider them as part of the framework. Their syntax and conventions, described in this chapter, were adapted so that they differ from the symfony ones as little as possible.



In a symfony project, all the applications share the same model. That's the whole point of the project level: regrouping applications that rely on common business rules. This is the reason that the model is independent from the applications and the model files are stored in a `lib/model/` directory at the root of the project.

Symfony's Database Schema

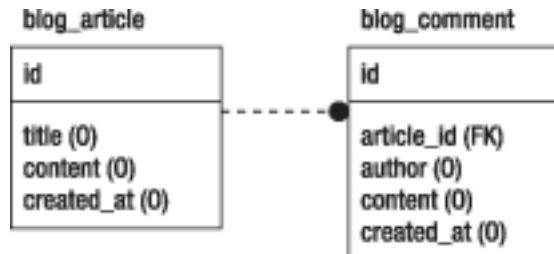
In order to create the data object model that symfony will use, you need to translate whatever relational model your database has to an object data model. The ORM needs a description of the relational model to do the mapping, and this is called a schema. In a schema, you define the tables, their relations, and the characteristics of their columns.

Symfony's syntax for schemas uses the YAML format. The `schema.yml` files must be located in the `myproject/config/doctrine` directory.

Schema Example

How do you translate a database structure into a schema? An example is the best way to understand it. Imagine that you have a blog database with two tables: `blog_article` and `blog_comment`, with the structure shown in Figure 8-1.

Figure 8-1 - A blog database table structure



The related `schema.yml` file should look like Listing 8-3.

Listing 8-3 - Sample schema.yml

```

Article:
  actAs: [Timestampable]
  tableName: blog_article
  columns:
    id:
      type: integer
      primary: true
      autoincrement: true
    title: string(255)
    content: clob

Comment:
  actAs: [Timestampable]
  tableName: blog_comment
  columns:
    id:
      type: integer
      primary: true
      autoincrement: true
    article_id: integer
    author: string(255)
    content: clob
  relations:
    Article:
      onDelete: CASCADE
      foreignAlias: Comments
  
```

*Listing
8-3*

Notice that the name of the database itself (`blog`) doesn't appear in the `schema.yml` file. Instead, the database is described under a connection name (`doctrine` in this example). This is because the actual connection settings can depend on the environment in which your application runs. For instance, when you run your application in the development environment, you will access a development database (maybe `blog_dev`), but with the same schema as the production database. The connection settings will be specified in the `databases.yml` file, described in the "Database Connections" section later in this chapter.

The schema doesn't contain any detailed connection to settings, only a connection name, to maintain database abstraction.

Basic Schema Syntax

In a `schema.yml` file, the first key represents a model name. You can specify multiple models, 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 model can have special attributes, including the `tableName` (the name of the models database table). If you don't mention a `tableName` for a model, Doctrine creates it based on the underscored version of the model name.



The underscore convention adds underscores between words, and lowerscases everything. The default underscored versions of `Article` and `Comment` are `article` and `comment`.

A model contains columns. The column value can be defined in two different ways:

- If you define only one attribute, it is the column type. Symfony understands the usual column types: `boolean`, `integer`, `float`, `date`, `string(size)`, `clob` (converted, for instance, to `text` in MySQL), and so on.
- If you need to define other column attributes (like default value, required, and so on), you should write the column attributes as a set of `key: value`. This extended schema syntax is described later in the chapter.

Models can also contain explicit foreign keys and indexes. Refer to the "Extended Schema Syntax" section later in this chapter to learn more.

Model Classes

The schema is used to build the model classes of the ORM layer. To save execution time, these classes are generated with a command-line task called `doctrine:build-model`.

*Listing
8-4*

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



After building your model, you must remember to clear symfony's internal cache with `php symfony cc` so symfony can find your newly created models.

Typing this command will launch the analysis of the schema and the generation of base data model classes in the `lib/model/doctrine/base` directory of your project:

- `BaseArticle.php`
- `BaseComment.php`

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

- `Article.php`
- `ArticleTable.php`
- `Comment.php`
- `CommentTable.php`

You defined only two models, and you end up with six files. There is nothing wrong, but it deserves some explanation.

Base and Custom Classes

Why keep two versions of the data object model in two different directories?

You will probably need to add custom methods and properties to the model objects (think about the `getName()` method in Listing 8-1). But as your project develops, you will also add tables or columns. Whenever you change the `schema.yml` file, you need to regenerate the object model classes by making a new call to `doctrine:build-model`. If your custom methods were written in the classes actually generated, they would be erased after each generation.

The `Base` classes kept in the `lib/model/doctrine/base/` directory are the ones directly generated from the schema. You should never modify them, since every new build of the model will completely erase these files.

On the other hand, the custom object classes, kept in the `lib/model/doctrine` directory, actually inherit from the `Base` ones. When the `doctrine:build-model` task is called on an existing model, these classes are not modified. So this is where you can add custom methods.

Listing 8-4 presents an example of a custom model class as created by the first call to the `doctrine:build-model` task.

Listing 8-4 - Sample Model Class File, in lib/model/doctrine/Article.php

```
class Article extends BaseArticle
{
}
```

*Listing
8-5*

It inherits everything of the `BaseArticle` class, but a modification in the schema will not affect it.

The mechanism of custom classes extending base classes allows you to start coding, even without knowing the final relational model of your database. The related file structure makes the model both customizable and evolutionary.

Object and Table Classes

`Article` and `Comment` are object classes that represent a record in the database. They give access to the columns of a record and to related records. This means that you will be able to know the title of an article by calling a method of an `Article` object, as in the example shown in Listing 8-5.

Listing 8-5 - Getters for Record Columns Are Available in the Object Class

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

*Listing
8-6*

`ArticleTable` and `CommentTable` are table classes; that is, classes that contain public methods to operate on the tables. They provide a way to retrieve records from the tables. Their methods usually return an object or a collection of objects of the related object class, as shown in Listing 8-6.

Listing 8-6 - Public Methods to Retrieve Records Are Available in the Table Class

```
// $article is an instance of class Article
$article = Doctrine_Core::getTable('Article')->find(123);
```

*Listing
8-7*

Accessing Data

In symfony, your data is accessed through objects. If you are used to the relational model and using SQL to retrieve and alter your data, the object model methods will likely look complicated. But once you've tasted the power of object orientation for data access, you will probably like it a lot.

But first, let's make sure we share the same vocabulary. Relational and object data model use similar concepts, but they each have their own nomenclature:

Relational	Object-Oriented
Table	Class
Row, record	Object
Field, column	Property

Retrieving the Column Value

When symfony builds the model, it creates one base object class for each of the models defined in the `schema.yml`. Each of these classes comes with default accessors and mutators based on the column definitions: The `new`, `getXXX()`, and `setXXX()` methods help to create objects and give access to the object properties, as shown in Listing 8-7.

Listing 8-7 - Generated Object Class Methods

```
Listing 8-8
$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();
```



The generated object class is called `Article` but is stored in a table named `blog_article` in the database. If the `tableName` were not defined in the schema, the class would have been called `article`. The accessors and mutators use a camelCase variant of the column names, so the `getTitle()` method retrieves the value of the `title` column.

To set several fields at one time, you can use the `fromArray()` method, also available for each object class, as shown in Listing 8-8.

Listing 8-8 - The fromArray() Method Is a Multiple Setter

```
Listing 8-9
$article->fromArray(
    'Title' => 'My first article',
    'Content' => 'This is my very first article.\n Hope you enjoy it!',
);
```

Retrieving Related Records

The `article_id` column in the `blog_comment` table implicitly defines a foreign key to the `blog_article` table. Each comment is related to one article, and one article can have many comments. The generated classes contain five methods translating this relationship in an object-oriented way, as follows:

- `$comment->getArticle()`: To get the related Article object
- `$comment->getArticleId()`: To get the ID of the related Article object
- `$comment->setArticle($article)`: To define the related Article object
- `$comment->setArticleId($id)`: To define the related Article object from an ID
- `$article->getComments()`: To get the related Comment objects

The `getArticleId()` and `setArticleId()` methods show that you can consider the `article_id` column as a regular column and set the relationships by hand, but they are not very interesting. The benefit of the object-oriented approach is much more apparent in the three other methods. Listing 8-9 shows how to use the generated setters.

Listing 8-9 - Foreign Keys Are Translated into a Special Setter

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

// Attach this comment to the previous $article object
$comment->setArticle($article);

// Alternative syntax
// Only makes sense if the object is already saved in the database
$comment->setArticleId($article->getId());
```

*Listing
8-10*

Listing 8-10 shows how to use the generated getters. It also demonstrates how to chain method calls on model objects.

Listing 8-10 - Foreign Keys Are Translated into Special Getters

```
// Many to one relationship
echo $comment->getArticle()->getTitle();
=> My first article
echo $comment->getArticle()->getContent();
=> This is my very first article.
    Hope you enjoy it!

// One to many relationship
$comments = $article->getComments();
```

*Listing
8-11*

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).

The `$comments` variable in Listing 8-10 contains an array of objects of class `Comment`. You can display the first one with `$comments[0]` or iterate through the collection with `foreach ($comments as $comment)`.

Saving and Deleting Data

By calling the new constructor, you created a new object, but not an actual record in the `blog_article` table. Modifying the object has no effect on the database either. In order to save the data into the database, you need to call the `save()` method of the object.

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

*Listing
8-12*

The ORM is smart enough to detect relationships between objects, so saving the `$article` object also saves the related `$comment` object. It also knows if the saved object has an existing counterpart in the database, so the call to `save()` is sometimes translated in SQL by

an `INSERT`, and sometimes by an `UPDATE`. The primary key is automatically set by the `save()` method, so after saving, you can retrieve the new primary key with `$article->getId()`.



You can check if an object is new by calling `isNew()`. And if you wonder if an object has been modified and deserves saving, call its `isModified()` method.

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, as shown in Listing 8-11.

Listing 8-11 - Delete Records from the Database with the `delete()` Method on the Related Object

```
Listing 8-13   foreach ($article->getComments() as $comment)
{           $comment->delete();
}
```

Retrieving Records by Primary Key

If you know the primary key of a particular record, use the `find()` class method of the table class to get the related object.

```
Listing 8-14   $article = Doctrine_Core::getTable('Article')->find(7);
```

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

In some cases, a primary key may consist of more than one column. In those cases, the `find()` method accepts multiple parameters, one for each primary key column.

Retrieving Records with Doctrine_Query

When you want to retrieve more than one record, you need to call the `createQuery()` method of the table class corresponding to the objects you want to retrieve. For instance, to retrieve objects of class `Article`, call `Doctrine_Core::getTable('Article')->createQuery()->execute()`.

The first parameter of the `execute()` method is an array of parameters, which is the array of values to replace any placeholders found in your query.

An empty `Doctrine_Query` returns all the objects of the class. For instance, the code shown in Listing 8-12 retrieves all the articles.

Listing 8-12 - Retrieving Records by Doctrine_Query with `createQuery()`—Empty Query

```
Listing 8-15   $q = Doctrine_Core::getTable('Article')->createQuery();
$q->execute();

// Will result in the following SQL query
SELECT b.id AS b_id, b.title AS b_title, b.content AS b_content,
b.created_at AS b_created_at, b.updated_at AS b_updated_at FROM
blog_article b
```

Hydrating

The call to `->execute()` is actually much more powerful than a simple SQL query. First, the SQL is optimized for the DBMS you choose. Second, any value passed to the `Doctrine_Query` is escaped before being integrated into the SQL code, which prevents SQL injection risks. Third, the method returns an array of objects, rather than a result set. The ORM automatically creates and populates objects based on the database result set. This process is called hydrating.

For a more complex object selection, you need an equivalent of the WHERE, ORDER BY, GROUP BY, and other SQL statements. The `Doctrine_Query` object has methods and parameters for all these conditions. For example, to get all comments written by Steve, ordered by date, build a `Doctrine_Query` as shown in Listing 8-13.

Listing 8-13 - Retrieving Records by a Doctrine_Query with createQuery()—Doctrine_Query with Conditions

```
$q = Doctrine_Core::getTable('Comment')
->createQuery('c')
->where('c.author = ?', 'Steve')
->orderBy('c.created_at ASC');
$comments = $q->execute();

// Will result in the following SQL query
SELECT b.id AS b_id, b.article_id AS b_article_id, b.author AS
b_author, b.content AS b_content, b.created_at AS b_created_at,
b.updated_at AS b_updated_at FROM blog_comment b WHERE (b.author = ?)
ORDER BY b.created_at ASC
```

Listing 8-16

Table 8-1 compares the SQL syntax with the `Doctrine_Query` object syntax.

Table 8-1 - SQL and Criteria Object Syntax

SQL	Criteria
WHERE column = value	<code>->where('acolumn = ?', 'value')</code>
Other SQL Keywords	
ORDER BY column ASC	<code>->orderBy('acolumn ASC')</code>
ORDER BY column DESC	<code>->addOrderBy('acolumn DESC')</code>
LIMIT limit	<code>->limit(limit)</code>
OFFSET offset	<code>->offset(offset)</code>
FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2	<code>->leftJoin('a.Model2 m')</code>
FROM table1 INNER JOIN table2 ON table1.col1 = table2.col2	<code>->innerJoin('a.Model2 m')</code>

Listing 8-14 shows another example of `Doctrine_Query` with multiple conditions. It retrieves all the comments by Steve on articles containing the word “enjoy,” ordered by date.

Listing 8-14 - Another Example of Retrieving Records by Doctrine_Query with createQuery()—Doctrine_Query with Conditions

Listing 8-17

```
$q = Doctrine_Core::getTable('Comment')
->createQuery('c')
->where('c.author = ?', 'Steve')
->leftJoin('c.Article a')
->andWhere('a.content LIKE ?', '%enjoy%')
->orderBy('c.created_at ASC');
$comments = $q->execute();

// Will result in the following SQL query
SELECT b.id AS b_id, b.article_id AS b_article_id, b.author AS
b_author, b.content AS b_content, b.created_at AS b_created_at,
b.updated_at AS b_updated_at, b2.id AS b2_id, b2.title AS b2_title,
b2.content AS b2_content, b2.created_at AS b2_created_at, b2.updated_at
AS b2_updated_at FROM blog_comment b LEFT JOIN blog_article b2 ON
b.article_id = b2.id WHERE (b.author = ? AND b2.content LIKE ?) ORDER BY
b.created_at ASC
```

Just as SQL is a simple language that allows you to build very complex queries, the `Doctrine_Query` object can handle conditions with any level of complexity. But since many developers think first in SQL before translating a condition into object-oriented logic, the `Doctrine_Query` object may be difficult to comprehend at first. The best way to understand it is to learn from examples and sample applications. The symfony project website, for instance, is full of `Doctrine_Query` building examples that will enlighten you in many ways.

Every `Doctrine_Query` instance has a `count()` method, which simply counts the number of records for the query and returns an integer. As there is no object to return, the hydrating process doesn't occur in this case, and the `count()` method is faster than `execute()`.

The table classes also provide `findAll()`, `findBy*`(`), and findOneBy*() methods, which are shortcuts for constructing Doctrine_Query instances, executing them and returning the results.`

Finally, if you just want the first object returned, replace `execute()` with a `fetchOne()` call. This may be the case when you know that a `Doctrine_Query` will return only one result, and the advantage is that this method returns an object rather than an array of objects.



When a `execute()` query returns a large number of results, you might want to display only a subset of it in your response. Symfony provides a pager class called `sDoctrinePager`, which automates the pagination of results.

Using Raw SQL Queries

Sometimes, you don't want to retrieve objects, but want to get only synthetic results calculated by the database. For instance, to get the latest creation date of all articles, it doesn't make sense to retrieve all the articles and to loop on the array. You will prefer to ask the database to return only the result, because it will skip the object hydrating process.

On the other hand, you don't want to call the PHP commands for database management directly, because then you would lose the benefit of database abstraction. This means that you need to bypass the ORM (Doctrine) but not the database abstraction (PDO).

Querying the database with PHP Data Objects requires that you do the following:

1. Get a database connection.
2. Build a query string.
3. Create a statement out of it.
4. Iterate on the result set that results from the statement execution.

If this looks like gibberish to you, the code in Listing 8-15 will probably be more explicit.

Listing 8-15 - Custom SQL Query with PDO

```
$connection = Doctrine_Manager::connection();
$query = 'SELECT MAX(created_at) AS max FROM blog_article';
$statement = $connection->execute($query);
$statement->execute();
$resultset = $statement->fetch(PDO::FETCH_OBJ);
$max = $resultset->max;
```

Listing
8-18

Just like Doctrine selections, PDO queries are tricky when you first start using them. Once again, examples from existing applications and tutorials will show you the right way.



If you are tempted to bypass this process and access the database directly, you risk losing the security and abstraction provided by Doctrine. Doing it the Doctrine way is longer, but it forces you to use good practices that guarantee the performance, portability, and security of your application. This is especially true for queries that contain parameters coming from a untrusted source (such as an Internet user). Doctrine does all the necessary escaping and secures your database. Accessing the database directly puts you at risk of SQL-injection attacks.

Using Special Date Columns

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

The good news is that Doctrine has a `Timestampable` behavior that will handle their updates for you. You don't need to manually set the `created_at` and `updated_at` columns; they will automatically be updated, as shown in Listing 8-16.

Listing 8-16 - `created_at` and `updated_at` Columns Are Dealt with Automatically

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->save();

// Show the creation date
echo $comment->getCreatedAt();
=> [date of the database INSERT operation]
```

Listing
8-19

Refactoring to the Data layer

When developing a symfony project, you often start by writing the domain logic code in the actions. But the database queries and model manipulation should not be stored in the controller layer. So 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 short and readable.

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

```
Listing 8-20 public function executeShowPopularArticlesForTag($request)
{
    $tag =
    Doctrine_Core::getTable('Tag')->findOneByName($request->getParameter('tag'));
    $this->forward404Unless($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. It makes the action more readable, and the model code can easily be reused in another action.

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 contain more than ten lines of PHP code.

Database Connections

The data model is independent from the database used, but you will definitely use a database. The minimum information required by symfony to send requests to the project database is the name, the credentials, and the type of database. These connection settings can be configured by passing a data source name (DSN) to the `configure:database` task:

```
Listing 8-21 $ php symfony configure:database "mysql:host=localhost;dbname=blog" root mYsEcret
```

The connection settings are environment-dependent. You can define distinct settings for the `prod`, `dev`, and `test` environments, or any other environment in your application by using the `env` option:

```
Listing 8-22 $ php symfony configure:database --env=dev
"mysql:host=localhost;dbname=blog_dev" root mYsEcret
```

This configuration can also be overridden per application. For instance, you can use this approach to have different security policies for a front-end and a back-end application, and define several database users with different privileges in your database to handle this:

```
Listing 8-23 $ php symfony configure:database --app=frontend
"mysql:host=localhost;dbname=blog" root mYsEcret
```

For each environment, you can define many connections. The default connection name used is `doctrine`. The `name` option allows you to create another connection:

```
$ php symfony configure:database --name=main
"mysql:host=localhost;dbname=example" root mYsEcret
```

Listing 8-24

You can also enter these connection settings manually in the `databases.yml` file located in the `config/` directory. Listing 8-17 shows an example of such a file and Listing 8-18 shows the same example with the extended notation.

Listing 8-17 - Shorthand Database Connection Settings

```
all:
  doctrine:
    class:      sfDoctrineDatabase
    param:
      dsn:       mysql://login:passwd@localhost/blog
```

Listing 8-25

Listing 8-18 - Sample Database Connection Settings, in myproject/config/databases.yml

```
prod:
  doctrine:
    param:
      dsn:       mysql:dbname=blog;host=localhost
      username:  login
      password:  passwd
    attributes:
      quote_identifier: false
      use_native_enum:  false
      validate: all
      idxname_format:  %s_idx
      seqname_format:  %s_seq
      tblname_format:  %s
```

Listing 8-26

To override the configuration per application, you need to edit an application-specific file, such as `apps/frontend/config/databases.yml`.

If you want to use a SQLite database, the `dsn` parameter must be set to the path of the database file. For instance, if you keep your blog database in `data/blog.db`, the `databases.yml` file will look like Listing 8-19.

Listing 8-19 - Database Connection Settings for SQLite Use a File Path As Host

```
all:
  doctrine:
    class:      sfDoctrineDatabase
    param:
      dsn:       sqlite:///SF_DATA_DIR%/blog.db
```

Listing 8-27

Extending the Model

The generated model methods are great but often not sufficient. As soon as you implement your own business logic, you need to extend it, either by adding new methods or by overriding existing ones.

Adding New Methods

You can add new methods to the empty model classes generated in the `lib/model/doctrine` directory. Use `$this` to call methods of the current object, and use `self::` to call static methods of the current class. Remember that the custom classes inherit methods from the `Base` classes located in the `lib/model/doctrine/base` directory.

For instance, for the `Article` object generated based on Listing 8-3, you can add a magic `__toString()` method so that echoing an object of class `Article` displays its title, as shown in Listing 8-20.

Listing 8-20 - Customizing the Model, in lib/model/doctrine/Article.php

```
Listing 8-28
class Article extends BaseArticle
{
    public function __toString()
    {
        return $this->getTitle(); // getTitle() is inherited from BaseArticle
    }
}
```

You can also extend the table classes—for instance, to add a method to retrieve all articles ordered by creation date, as shown in Listing 8-21.

Listing 8-21 - Customizing the Model, in lib/model/doctrine/ArticleTable.php

```
Listing 8-29
class ArticleTable extends BaseArticleTable
{
    public function getAllOrderedByDate()
    {
        $q = $this->createQuery('a')
            ->orderBy('a.created_at ASC');

        return $q->execute();
    }
}
```

The new methods are available in the same way as the generated ones, as shown in Listing 8-22.

Listing 8-22 - Using Custom Model Methods Is Like Using the Generated Methods

```
Listing 8-30
$articles = Doctrine_Core::getTable('Article')->getAllOrderedByDate();
foreach ($articles as $article)
{
    echo $article; // Will call the magic __toString() method
}
```

Overriding Existing Methods

If some of the generated methods in the `Base` classes don't fit your requirements, you can still override them in the custom classes. Just make sure that you use the same method signature (that is, the same number of arguments).

For instance, the `$article->getComments()` method returns a collection of `Comment` objects, in no particular order. If you want to have the results ordered by creation date, with the latest comment coming first, then create the `getComments()` method, as shown in Listing 8-23.

Listing 8-23 - Overriding Existing Model Methods, in lib/model/doctrine/Article.php

```
public function getComments()
{
    $q = Doctrine_Core::getTable('Comment')
        ->createQuery('c')
        ->where('c.article_id = ?', $this->getId())
        ->orderBy('c.created_at ASC');

    return $q->execute();
}
```

*Listing
8-31*

Using Model Behaviors

Some model modifications are generic and can be reused. For instance, methods to make a model object sortable and an optimistic lock to prevent conflicts between concurrent object saving are generic extensions that can be added to many classes.

Symfony packages these extensions into behaviors. Behaviors are external classes that provide additional methods to model classes. The model classes already contain hooks, and symfony knows how to extend them.

To enable behaviors in your model classes, you must modify your schema and use the `actAs` option:

```
Article:
    actAs: [Timestampable, Sluggable]
    tableName: blog_article
    columns:
        id:
            type: integer
            primary: true
            autoincrement: true
        title: string(255)
        content: clob
```

*Listing
8-32*

After rebuilding the model, the `Article` model have a `slug` column that is automatically set to a url friendly string based on the title.

Some behaviors that come with Doctrine are:

- `Timestampable`
- `Sluggable`
- `SoftDelete`
- `Searchable`
- `I18n`
- `Versionable`
- `NestedSet`

Extended Schema Syntax

A `schema.yml` file can be simple, as shown in Listing 8-3. But relational models are often complex. That's why the schema has an extensive syntax able to handle almost every case.

Attributes

Connections and tables can have specific attributes, as shown in Listing 8-24. They are set under an `_attributes` key.

Listing 8-24 - Attributes for Model Settings

Listing 8-33

```
Article:  
  attributes:  
    export: tables  
    validate: none
```

The `export` attribute controls what SQL is exported to the database when creating tables for that model. Using the `tables` value would only export the table structure and no foreign keys, indexes, etc.

Tables that contain localized content (that is, several versions of the content, in a related table, for internationalization) should use the `I18n` behavior (see Chapter 13 for details), as shown in Listing 8-25.

Listing 8-25 - I18n Behavior

Listing 8-34

```
Article:  
  actAs:  
    I18n:  
      fields: [title, content]
```

Dealing with multiple Schemas

You can have more than one schema per application. Symfony will take into account every file ending with `.yml` in the `config/doctrine` folder. If your application has many models, or if some models don't share the same connection, you will find this approach very useful.

Consider these two schemas:

```
// In config/doctrine/business-schema.yml
Article:
    id:
        type: integer
        primary: true
        autoincrement: true
    title: string(50)

// In config/doctrine/stats-schema.yml
Hit:
    actAs: [Timestampable]
    columns:
        id:
            type: integer
            primary: true
            autoincrement: true
        resource: string(100)
```

*Listing
8-35*

Both schemas share the same connection (`doctrine`), and the `Article` and `Hit` classes will be generated under the same `lib/model/doctrine` directory. Everything happens as if you had written only one schema.

You can also have different schemas use different connections (for instance, `doctrine` and `doctrine_bis`, to be defined in `databases.yml`) and associate it with that connection:

```
// In config/doctrine/business-schema.yml
Article:
    connection: doctrine
    id:
        type: integer
        primary: true
        autoincrement: true
    title: string(50)

// In config/doctrine/stats-schema.yml
Hit:
    connection: doctrine_bis
    actAs: [Timestampable]
    columns:
        id:
            type: integer
            primary: true
            autoincrement: true
        resource: string(100)
```

*Listing
8-36*

Many applications use more than one schema. In particular, some plug-ins have their own schema to avoid messing with your own classes (see Chapter 17 for details).

Column Details

The basic syntax lets you define the type with one of the type keywords. Listing 8-26 demonstrates these choices.

Listing 8-26 - Basic Column Attributes

```
Listing 8-37 Article:
columns:
    title: string(50) # Specify the type and length
```

But you can define much more for a column. If you do, you will need to define column settings as an associative array, as shown in Listing 8-27.

Listing 8-27 - Complex Column Attributes

```
Listing 8-38 Article:
columns:
    id: { type: integer, notnull: true, primary: true,
autoincrement: true }
    name: { type: string(50), default: foobar }
    group_id: { type: integer }
```

The column parameters are as follows:

- **type**: Column type. The choices are `boolean`, `integer`, `double`, `float`, `decimal`, `string(size)`, `date`, `time`, `timestamp`, `blob`, and `clob`.
- **notnull**: Boolean. Set it to `true` if you want the column to be required.
- **length**: The size or length of the field for types that support it
- **scale**: Number of decimal places for use with decimal data type (size must also be specified)
- **default**: Default value.
- **primary**: Boolean. Set it to `true` for primary keys.
- **autoincrement**: Boolean. Set it to `true` for columns of type `integer` that need to take an auto-incremented value.
- **sequence**: Sequence name for databases using sequences for `autoIncrement` columns (for example, PostgreSQL and Oracle).
- **unique**: Boolean. Set it to `true` if you want the column to be unique.

Relationships

You can specify foreign key relationships under the `relations` key in a model. The schema in Listing 8-28 will create a foreign key on the `user_id` column, matching the `id` column in the `blog_user` table.

Listing 8-28 - Foreign Key Alternative Syntax

```
Listing 8-39 Article:
actAs: [Timestampable]
tableName: blog_article
columns:
    id:
        type: integer
        primary: true
        autoincrement: true
        title: string(255)
        content: clob
        user_id: integer
relations:
```

```
User:
onDelete: CASCADE
foreignAlias: Articles
```

Indexes

You can add indexes under the `indexes:` key in a model. If you want to define unique indexes, you must use the `type: unique` syntax. For columns that require a size, because they are text columns, the size of the index is specified the same way as the length of the column using parentheses. Listing 8-30 shows the alternative syntax for indexes.

Listing 8-30 - Indexes and Unique Indexes

```
Article:
  actAs: [Timestampable]
  tableName: blog_article
  columns:
    id:
      type: integer
      primary: true
      autoincrement: true
    title: string(255)
    content: clob
    user_id: integer
  relations:
    User:
      onDelete: CASCADE
      foreignAlias: Articles
  indexes:
    my_index:
      fields:
        title:
          length: 10
        user_id: []
    my_other_index:
      type: unique
      fields:
        created_at
```

*Listing
8-40*

i18n Tables

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

Listing 8-33 - Explicit i18n Mechanism

```
DbGroup:
  actAs:
    I18n:
      fields: [name]
  columns:
    name: string(50)
```

*Listing
8-41*

Behaviors

Behaviors are model modifiers provided by plug-ins that add new capabilities to your Doctrine classes. Chapter 17 explains more about behaviors. You can define behaviors right in the

schema, by listing them for each table, together with their parameters, under the `actAs` key. Listing 8-34 gives an example by extending the `Article` class with the `Sluggable` behavior.

Listing 8-34 - Behaviors Declaration

```
Listing 8-42 Article:  
actAs: [Sluggable]  
# ...
```

Don't Create the Model Twice

The trade-off of using an ORM is that you must define the data structure twice: once for the database, and once for the object model. Fortunately, symfony offers command-line tools to generate one based on the other, so you can avoid duplicate work.

Building a SQL Database Structure Based on an Existing Schema

If you start your application by writing the `schema.yml` file, symfony can generate a SQL query that creates the tables directly from the YAML data model. To use the query, go to your root project directory and type this:

```
Listing 8-43 $ php symfony doctrine:build-sql
```

A `schema.sql` file will be created in `myproject/data/sql/`. Note that the generated SQL code will be optimized for the database system defined in the `databases.yml`.

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

```
Listing 8-44 $ mysqladmin -u root -p create blog  
$ mysql -u root -p blog < data/sql/schema.sql
```

The generated SQL is also helpful to rebuild the database in another environment, or to change to another DBMS.



The command line also offers a task to populate your database with data based on a text file. See Chapter 16 for more information about the `doctrine:dat-load` task and the YAML fixture files.

Generating a YAML Data Model from an Existing Database

Symfony can use Doctrine to generate a `schema.yml` file from an existing database, thanks to introspection (the capability of databases to determine the structure of the tables on which they are operating). 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 need to make sure that the project `databases.yml` file points to the correct database and contains all connection settings, and then call the `doctrine:build-schema` command:

```
Listing 8-45 $ php symfony doctrine:build-schema
```

A brand-new `schema.yml` file built from your database structure is generated in the `config/doctrine/` directory. You can build your model based on this schema.

Summary

Symfony uses Doctrine as the ORM and PHP Data Objects as the database abstraction layer. It means that you must first describe the relational schema of your database in YAML before generating the object model classes. Then, at runtime, use the methods of the object and table classes to retrieve information about a record or a set of records. You can override them and extend the model easily by adding methods to the custom classes. The connection settings are defined in a `databases.yml` file, which can support more than one connection. And the command line contains special tasks to avoid duplicate structure definition.

The model layer is the most complex of the symfony framework. One reason for this complexity is that data manipulation is an intricate matter. The related security issues are crucial for a website and should not be ignored. Another reason is that symfony is more suited for middle- to large-scale applications in an enterprise context. In such applications, the automations provided by the symfony model really represent a gain of time, worth the investment in learning its internals.

So don't hesitate to spend some time testing the model objects and methods to fully understand them. The solidity and scalability of your applications will be a great reward.

Chapter 9

Links And The Routing System

Links and URLs deserve particular treatment in a web application framework. This is because the unique entry point of the application (the front controller) and the use of helpers in templates allow for a complete separation between the way URLs work and their appearance. This is called routing. More than a gadget, routing is a useful tool to make web applications even more user-friendly and secure. This chapter will tell you everything you need to know to handle URLs in your symfony applications:

- What the routing system is and how it works
- How to use link helpers in templates to enable routing of outgoing URLs
- How to configure the routing rules to change the appearance of URLs

You will also find a few tricks for mastering routing performance and adding finishing touches.

What Is Routing?

Routing is a mechanism that rewrites URLs to make them more user-friendly. But to understand why this is important, you must first take a few minutes to think about URLs.

URLs As Server Instructions

URLs carry information from the browser to the server required to enact an action as desired by the user. For instance, a traditional URL contains the file path to a script and some parameters necessary to complete the request, as in this example: `http://www.example.com/web/controller/article.php?id=123456&format_code=6532`

This URL conveys information about the application's architecture and database. Developers usually hide the application's infrastructure in the interface (for instance, they choose page titles like "Personal profile page" rather than "QZ7.65"). Revealing vital clues to the internals of the application in the URL contradicts this effort and has serious drawbacks:

- The technical data appearing in the URL creates potential security breaches. In the preceding example, what happens if an ill-disposed user changes the value of the `id` parameter? Does this mean the application offers a direct interface to the database? Or what if the user tries other script names, like `admin.php`, just for fun? All in all, raw URLs offer an easy way to hack an application, and managing security is almost impossible with them.
- The unintelligibility of URLs makes them disturbing wherever they appear, and they dilute the impact of the surrounding content. And nowadays, URLs don't appear only in the address bar. They appear when a user hovers the mouse over a link, as well as in search results. When users look for information, you want to give them

easily understandable clues regarding what they found, rather than a confusing URL such as the one shown in Figure 9-1.

Figure 9-1 - URLs appear in many places, such as in search results

Microsoft Office Clip Art and Media Home Page

Microsoft Office Clip Art and Media - Over 140000 clip art graphics, animations, photos, and sounds for use in **Microsoft** Office products.
office.microsoft.com/clipart/default.aspx?lc=en-us - 56k - 30 Aug 2006 -
[Cached](#) - [Similar pages](#)

- If one URL has to be changed (for instance, if a script name or one of its parameters is modified), every link to this URL must be changed as well. It means that modifications in the controller structure are heavyweight and expensive, which is not ideal in agile development.

And it could be much worse if symfony didn't use the front controller paradigm; that is, if the application contained many scripts accessible from the Internet, in many directories, such as these:

`http://www.example.com/web/gallery/album.php?name=my%20holidays`
`http://www.example.com/web/weblog/public/post/list.php`
`http://www.example.com/web/general/content/page.php?name=about%20us`

Listing 9-1

In this case, developers would need to match the URL structure with the file structure, resulting in a maintenance nightmare when either structure changed.

URLs As Part of the Interface

The idea behind routing is to consider the URL as part of the interface. The application can format a URL to bring information to the user, and the user can use the URL to access resources of the application.

This is possible in symfony applications, because the URL presented to the end user is unrelated to the server instruction needed to perform the request. Instead, it is related to the resource requested, and it can be formatted freely. For instance, symfony can understand the following URL and have it display the same page as the first URL shown in this chapter:

`http://www.example.com/articles/finance/2010/activity-breakdown.html`

Listing 9-2

The benefits are immense:

- URLs actually mean something, and they can help the users decide if the page behind a link contains what they expect. A link can contain additional details about the resource it returns. This is particularly useful for search engine results. Additionally, URLs sometimes appear without any mention of the page title (think about when you copy a URL in an e-mail message), and in this case, they must mean something on their own. See Figure 9-2 for an example of a user-friendly URL.
- The technical implementation is hidden to the users : they do not know which script is used, they cannot guess an `id` or similar parameter: your application is less vulnerable to a potential security breach. More, you can totally change what happens behind the scene without affecting their experience (they won't have a 404 error or a permanent redirect).

Figure 9-2 - URLs can convey additional information about a page, like the publication date

[symfony PHP5 framework » AJAX pagination made simple](#)

The **AJAX pagination** demo uses two very simple actions, both passing a pager to their template: class `pagerActions` extends `sfActions` { public function ...

www.symfony-project.com/weblog/2006/07/17/ajax-pagination-made-simple.html - 12k -
[Cached](#) - [Similar pages](#)

- URLs written in paper documents are easier to type and remember. If your company website appears as `http://www.example.com/controller/web/index.jsp?id=ERD4` on your business card, it will probably not receive many visits.
- The URL can become a command-line tool of its own, to perform actions or retrieve information in an intuitive way. Applications offering such a possibility are faster to use for power users.

Listing 9-3 // List of results: add a new tag to narrow the list of results
<http://del.icio.us/tag/symfony+ajax>
// User profile page: change the name to get another user profile
<http://www.askeet.com/user/francois>

- You can change the URL formatting and the action name/parameters independently, with a single modification. It means that you can develop first, and format the URLs afterwards, without totally messing up your application.
- Even when you reorganize the internals of an application, the URLs can remain the same for the outside world. It makes URLs persistent, which is a must because it allows bookmarking on dynamic pages.
- Search engines tend to skip dynamic pages (ending with `.php`, `.asp`, and so on) when they index websites. So you can format URLs to have search engines think they are browsing static content, even when they meet a dynamic page, thus resulting in better indexing of your application pages.
- It is safer. Any unrecognized URL will be redirected to a page specified by the developer, and users cannot browse the web root file structure by testing URLs. The actual script name called by the request, as well as its parameters, is hidden.

The correspondence between the URLs presented to the user and the actual script name and request parameters is achieved by a routing system, based on patterns that can be modified through configuration.



How about assets? Fortunately, the URLs of assets (images, style sheets, and JavaScript) don't appear much during browsing, so there is no real need for routing for those. In symfony, all assets are located under the `web/` directory, and their URL matches their location in the file system. However, you can manage dynamic assets (handled by actions) by using a generated URL inside the asset helper. For instance, to display a dynamically generated image, use `image_tag(url_for('captcha/image?key='.$key))`.

How It Works

Symfony disconnects the external URL and its internal URI. The correspondence between the two is made by the routing system. To make things easy, symfony uses a syntax for internal URIs very similar to the one of regular URLs. Listing 9-1 shows an example.

Listing 9-1 - External URL and Internal URI

Listing 9-4 // Internal URI syntax
`<module>/<action>[?param1=value1][¶m2=value2][¶m3=value3]...`

```
// Example internal URI, which never appears to the end user
article/permalink?year=2010&subject=finance&title=activity-breakdown

// Example external URL, which appears to the end user
http://www.example.com/articles/finance/2010/activity-breakdown.html
```

The routing system uses a special configuration file, called `routing.yml`, in which you can define routing rules. Consider the rule shown in Listing 9-2. It defines a pattern that looks like `articles/*/*/*` and names the pieces of content matching the wildcards.

Listing 9-2 - A Sample Routing Rule

```
article_by_title:
  url:    articles/:subject/:year/:title.html
  param: { module: article, action: permalink }
```

*Listing
9-5*

Every request sent to a symfony application is first analyzed by the routing system (which is simple because every request is handled by a single front controller). The routing system looks for a match between the request URL and the patterns defined in the routing rules. If a match is found, the named wildcards become request parameters and are merged with the ones defined in the `param:` key. See how it works in Listing 9-3.

Listing 9-3 - The Routing System Interprets Incoming Request URLs

```
// The user types (or clicks on) this external URL
http://www.example.com/articles/finance/2010/activity-breakdown.html

// The front controller sees that it matches the article_by_title rule
// The routing system creates the following request parameters
'module'  => 'article'
'action'   => 'permalink'
'subject'  => 'finance'
'year'     => '2010'
'title'    => 'activity-breakdown'
```

*Listing
9-6*

The request is then passed to the `permalink` action of the `article` module, which has all the required information in the request parameters to determine which article is to be shown.

But the mechanism also must work the other way around. For the application to show external URLs in its links, you must provide the routing system with enough data to determine which rule to apply to it. You also must not write hyperlinks directly with `<a>` tags—this would bypass routing completely—but with a special helper, as shown in Listing 9-4.

Listing 9-4 - The Routing System Formats Outgoing URLs in Templates

```
// The url_for() helper transforms an internal URI into an external URL
<a href="<?php echo url_for('article/
permalink?subject=finance&year=2010&title=activity-breakdown') ?>">click
here</a>

// The helper sees that the URI matches the article_by_title rule
// The routing system creates an external URL out of it
=> <a href="http://www.example.com/articles/finance/2010/
activity-breakdown.html">click here</a>

// The link_to() helper directly outputs a hyperlink
// and avoids mixing PHP with HTML
<?php echo link_to(
  'click here',
```

*Listing
9-7*

```
'article/permalink?subject=finance&year=2010&title=activity-breakdown'
) ?>

// Internally, link_to() will make a call to url_for() so the result is
the same
=> <a href="http://www.example.com/articles/finance/2010/
activity-breakdown.html">click here</a>
```

So routing is a two-way mechanism, and it works only if you use the `link_to()` helper to format all your links.

URL Rewriting

Before getting deeper into the routing system, one matter needs to be clarified. In the examples given in the previous section, there is no mention of the front controller (`index.php` or `frontend_dev.php`) in the internal URIs. The front controller, not the elements of the application, decides the environment. So all the links must be environment-independent, and the front controller name can never appear in internal URIs.

There is no script name in the examples of generated URLs either. This is because generated URLs don't contain any script name in the production environment by default. The `no_script_name` parameter of the `settings.yml` file precisely controls the appearance of the front controller name in generated URLs. Set it to `false`, as shown in Listing 9-5, and the URLs output by the link helpers will mention the front controller script name in every link.

Listing 9-5 - Showing the Front Controller Name in URLs, in `apps/frontend/config/settings.yml`

Listing 9-8

```
prod:
  .settings
    no_script_name: false
```

Now, the generated URLs will look like this:

Listing 9-9

```
http://www.example.com/index.php/articles/finance/2010/
activity-breakdown.html
```

In all environments except the production one, the `no_script_name` parameter is set to `false` by default. So when you browse your application in the development environment, for instance, the front controller name always appears in the URLs.

Listing 9-10

```
http://www.example.com/frontend_dev.php/articles/finance/2010/
activity-breakdown.html
```

In production, the `no_script_name` is set to `true`, so the URLs show only the routing information and are more user-friendly. No technical information appears.

Listing 9-11

```
http://www.example.com/articles/finance/2010/activity-breakdown.html
```

But how does the application know which front controller script to call? This is where URL rewriting comes in. The web server can be configured to call a given script when there is none in the URL.

In Apache, this is possible once you have the `mod_rewrite` extension activated. Every Symfony project comes with an `.htaccess` file, which adds `mod_rewrite` settings to your server configuration for the `web/` directory. The default content of this file is shown in Listing 9-6.

Listing 9-6 - Default Rewriting Rules for Apache, in myproject/web/.htaccess

```
<IfModule mod_rewrite.c>
    RewriteEngine On

    # uncomment the following line, if you are having trouble
    # getting no_script_name to work
    #RewriteBase /

    # we skip all files with .something
    #RewriteCond %{REQUEST_URI} \..+$
    #RewriteCond %{REQUEST_URI} !\.html$
    #RewriteRule .* - [L]

    # we check if the .html version is here (caching)
    RewriteRule ^$ index.html [QSA]
    RewriteRule ^([^.]+)$ $1.html [QSA]
    RewriteCond %{REQUEST_FILENAME} !-f

    # no, so we redirect to our front web controller
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

Listing
9-12

The web server inspects the shape of the URLs it receives. If the URL does not contain a suffix (commented out by default) and if there is no cached version of the page available (Chapter 12 covers caching), then the request is handed to `index.php`.

However, the `web/` directory of a symfony project is shared among all the applications and environments of the project. It means that there is usually more than one front controller in the `web` directory. For instance, a project having a `frontend` and a `backend` application, and a `dev` and `prod` environment, contains four front controller scripts in the `web/` directory:

```
index.php      // frontend in prod
frontend_dev.php // frontend in dev
backend.php     // backend in prod
backend_dev.php // backend in dev
```

Listing
9-13

The `mod_rewrite` settings can specify only one default script name. If you set `no_script_name` to `true` for all the applications and environments, all URLs will be interpreted as requests to the `frontend` application in the `prod` environment. This is why you can have only one application with one environment taking advantage of the URL rewriting for a given project.



There is a way to have more than one application with no script name. Just create subdirectories in the `web` root, and move the front controllers inside them. Change the path to the `ProjectConfiguration` file accordingly, and create the `.htaccess` URL rewriting configuration that you need for each application.

Link Helpers

Because of the routing system, you should use link helpers instead of regular `<a>` tags in your templates. Don't look at it as a hassle, but rather as an opportunity to keep your application clean and easy to maintain. Besides, link helpers offer a few very useful shortcuts that you don't want to miss.

Hyperlinks, Buttons, and Forms

You already know about the `link_to()` helper. It outputs an XHTML-compliant hyperlink, and it expects two parameters: the element that can be clicked and the internal URI of the resource to which it points. If, instead of a hyperlink, you want a button, use the `button_to()` helper. Forms also have a helper to manage the value of the `action` attribute. You will learn more about forms in the next chapter. Listing 9-7 shows some examples of link helpers.

Listing 9-7 - Link Helpers for <a>, <input>, and <form> Tags

```
Listing 9-14 // Hyperlink on a string
<?php echo link_to('my article', 'article/read?title=Finance_in_France') ?>
=> <a href="/routed/url/to/Finance_in_France">my article</a>

// Hyperlink on an image
<?php echo link_to(image_tag('read.gif'), 'article/
read?title=Finance_in_France') ?>
=> <a href="/routed/url/to/Finance_in_France"></a>

// Button tag
<?php echo button_to('my article', 'article/read?title=Finance_in_France') ?>
=> <input value="my article"
type="button" onclick="document.location.href='/routed/url/to/
Finance_in_France';" />

// Form tag
<?php echo form_tag('article/read?title=Finance_in_France') ?>
=> <form method="post" action="/routed/url/to/Finance_in_France" />
```

Link helpers can accept internal URIs as well as absolute URLs (starting with `http://`, and skipped by the routing system) and anchors. Note that in real-world applications, internal URIs are built with dynamic parameters. Listing 9-8 shows examples of all these cases.

Listing 9-8 - URLs Accepted by Link Helpers

```
Listing 9-15 // Internal URI
<?php echo link_to('my article', 'article/read?title=Finance_in_France') ?>
=> <a href="/routed/url/to/Finance_in_France">my article</a>

// Internal URI with dynamic parameters
<?php echo link_to('my article', 'article/
read?title='.$article->getTitle()) ?>

// Internal URI with anchors
<?php echo link_to('my article', 'article/
read?title=Finance_in_France#foo') ?>
=> <a href="/routed/url/to/Finance_in_France#foo">my article</a>

// Absolute URL
<?php echo link_to('my article', 'http://www.example.com/foobar.html') ?>
=> <a href="http://www.example.com/foobar.html">my article</a>
```

Link Helper Options

As explained in Chapter 7, helpers accept an additional options argument, which can be an associative array or a string. This is true for link helpers, too, as shown in Listing 9-9.

Listing 9-9 - Link Helpers Accept Additional Options

```
// Additional options as an associative array
<?php echo link_to('my article', 'article/read?title=Finance_in_France',
array(
    'class'  => 'foobar',
    'target' => '_blank'
)) ?>

// Additional options as a string (same result)
<?php echo link_to('my article', 'article/
read?title=Finance_in_France','class=foobar target=_blank') ?>
=> <a href="/routed/url/to/Finance_in_France" class="foobar"
target="_blank">my article</a>
```

Listing 9-16

You can also add one of the symfony-specific options for link helpers: `confirm` and `popup`. The first one displays a JavaScript confirmation dialog box when the link is clicked, and the second opens the link in a new window, as shown in Listing 9-10.

Listing 9-10 - 'confirm' and 'popup' Options for Link Helpers

```
<?php echo link_to('delete item', 'item/delete?id=123', 'confirm=Are you
sure?') ?>
=> <a onclick="return confirm('Are you sure?');"
      href="/routed/url/to/delete/123.html">delete item</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', 'popup=true')
?>
=> <a onclick="window.open(this.href);return false;"
      href="/fo_dev.php/shoppingCart/add/id/100.html">add to cart</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', array(
    'popup' => array('popupWindow', 'width=310,height=400,left=320,top=0')
)) ?>
=> <a onclick="window.open(this.href,'popupWindow',
    'width=310,height=400,left=320,top=0');return false;"
      href="/fo_dev.php/shoppingCart/add/id/100.html">add to cart</a>
```

Listing 9-17

These options can be combined.

Fake GET and POST Options

Sometimes web developers use GET requests to actually do a POST. For instance, consider the following URL:

http://www.example.com/index.php/shopping_cart/add/id/100

Listing 9-18

This request will change the data contained in the application, by adding an item to a shopping cart object, stored in the session or in a database. This URL can be bookmarked, cached, and indexed by search engines. Imagine all the nasty things that might happen to the database or to the metrics of a website using this technique. As a matter of fact, this request should be considered as a POST, because search engine robots do not do POST requests on indexing.

Symfony provides a way to transform a call to a `link_to()` or `button_to()` helper into an actual POST. Just add a `post=true` option, as shown in Listing 9-11.

Listing 9-11 - Making a Link Call a POST Request

```
Listing 9-19 <?php echo link_to('go to shopping cart', 'shoppingCart/add?id=100',
array('post' => true)) ?>
=> <a onclick="f = document.createElement('form');
document.body.appendChild(f);
f.method = 'POST'; f.action = this.href; f.submit();return
false;" href="/shoppingCart/add/id/100.html">go to shopping cart</a>
```

This `<a>` tag has an `href` attribute, and browsers without JavaScript support, such as search engine robots, will follow the link doing the default GET. So you must also restrict your action to respond only to the POST method, by adding something like the following at the beginning of the action:

```
Listing 9-20 $this->forward404Unless($this->getRequest()->isMethod('post'));
```

Just make sure you don't use this option on links located in forms, since it generates its own `<form>` tag.

It is a good habit to tag as POST the links that actually post data.

Forcing Request Parameters As GET Variables

According to your routing rules, variables passed as parameters to a `link_to()` are transformed into patterns. If no rule matches the internal URI in the `routing.yml` file, the default rule transforms `module/action?key=value` into `/module/action/key/value`, as shown in Listing 9-12.

Listing 9-12 - Default Routing Rule

```
Listing 9-21 <?php echo link_to('my article', 'article/read?title=Finance_in_France') ?>
=> <a href="/article/read/title/Finance_in_France">my article</a>
```

If you actually need to keep the GET syntax—to have request parameters passed under the `?key=value` form—you should put the variables that need to be forced outside the URL parameter, in the `query_string` option. As this would conflict also with an anchor in the URL, you have to put it into the `anchor` option instead of prepending it to the internal URI. All the link helpers accept these options, as demonstrated in Listing 9-13.

Listing 9-13 - Forcing GET Variables with the query_string Option

```
Listing 9-22 <?php echo link_to('my article', 'article/read', array(
    'query_string' => 'title=Finance_in_France',
    'anchor' => 'foo'
)) ?>
=> <a href="/article/read?title=Finance_in_France#foo">my article</a>
```

A URL with request parameters appearing as GET variables can be interpreted by a script on the client side, and by the `$_GET` and `$_REQUEST` variables on the server side.

Asset helpers

Chapter 7 introduced the asset helpers `image_tag()`, `stylesheet_tag()`, and `javascript_include_tag()`, which allow you to include an image, a style sheet, or a JavaScript file in the response. The paths to such assets are not processed by the routing system, because they link to resources that are actually located under the public web directory.

You don't need to mention a file extension for an asset. Symfony automatically adds `.png`, `.js`, or `.css` to an image, JavaScript, or style sheet helper call. Also, Symfony will automatically look for those assets in the `web/images/`, `web/js/`, and `web/css/` directories. Of course, if you want to include a specific file format or a file from a specific location, just use the full file name or the full file path as an argument.

To fix the size of an image, use the `size` attribute. It expects a width and a height in pixels, separated by an `x`.

```
<?php echo image_tag('test', 'size=100x20') ?>
=> <img href="/images/test.png" width="100" height="20"/>
```

Listing 9-23

If you want the asset inclusion to be done in the `<head>` section (for JavaScript files and style sheets), you should use the `use_stylesheet()` and `use_javascript()` helpers in your templates, instead of the `_tag()` versions in the layout. They add the asset to the response, and these assets are included before the `</head>` tag is sent to the browser.

Using Absolute Paths

The link and asset helpers generate relative paths by default. To force the output to absolute paths, set the `absolute` option to `true`, as shown in Listing 9-14. This technique is useful for inclusions of links in an e-mail message, RSS feed, or API response.

Listing 9-14 - Getting Absolute URLs Instead of Relative URLs

```
<?php echo url_for('article/read?title=Finance_in_France') ?>
=> '/routed/url/to/Finance_in_France'
<?php echo url_for('article/read?title=Finance_in_France', true) ?>
=> 'http://www.example.com/routed/url/to/Finance_in_France'

<?php echo link_to('finance', 'article/read?title=Finance_in_France') ?>
=> <a href="/routed/url/to/Finance_in_France">finance</a>
<?php echo link_to('finance', 'article/
read?title=Finance_in_France', 'absolute=true') ?>
=> <a href=" http://www.example.com/routed/url/to/
Finance_in_France">finance</a>

// The same goes for the asset helpers
<?php echo image_tag('test', 'absolute=true') ?>
<?php echo javascript_include_tag('myscript', 'absolute=true') ?>
```

Listing 9-24

The Mail helper

Nowadays, e-mail-harvesting robots prowl about the Web, and you can't display an e-mail address on a website without becoming a spam victim within days. This is why symfony provides a `mail_to()` helper.

The `mail_to()` helper takes two parameters: the actual e-mail address and the string that should be displayed. Additional options accept an `encode` parameter to output something pretty unreadable in HTML, which is understood by browsers but not by robots.

Listing 9-25

```
<?php echo mail_to('myaddress@mydomain.com', 'contact') ?>
=> <a href="mailto:myaddress@mydomain.com">contact</a>
<?php echo mail_to('myaddress@mydomain.com', 'contact', 'encode=true') ?>
=> <a href="ma... om">ct...
ess</a>
```

Encoded e-mail messages are composed of characters transformed by a random decimal and hexadecimal entity encoder. This trick stops most of the address-harvesting spambots for now, but be aware that the harvesting techniques evolve rapidly.

Routing Configuration

The routing system does two things:

- It interprets the external URL of incoming requests and transforms it into an internal URI, to determine the module/action and the request parameters.
- It formats the internal URIs used in links into external URLs (provided that you use the link helpers).

The conversion is based on a set of routing rules. These rules are stored in a `routing.yml` configuration file located in the application `config/` directory. Listing 9-15 shows the default routing rules, bundled with every symfony project.

Listing 9-15 - The Default Routing Rules, in frontend/config/routing.yml

Listing 9-26

```
# default rules
homepage:
    url:   /
    param: { module: default, action: index }

# generic rules
# please, remove them by adding more specific rules
default_index:
    url:   /:module
    param: { action: index }

default:
    url:   /:module/:action/*
```

Rules and Patterns

Routing rules are bijective associations between an external URL and an internal URI. A typical rule is made up of the following:

- A unique label, which is there for legibility and speed, and can be used by the link helpers

- A pattern to be matched (`url` key)
- An array of request parameter values (`param` key)

Patterns can contain wildcards (represented by an asterisk, `*`) and named wildcards (starting with a colon, `:`). A match to a named wildcard becomes a request parameter value. For instance, the `default` rule defined in Listing 9-15 will match any URL like `/foo/bar`, and set the `module` parameter to `foo` and the `action` parameter to `bar`.



Named wildcards can be separated by a slash or a dot, so you can write a pattern like:

```
my_rule:
  url: /foo/:bar.:format
  param: { module: mymodule, action: myaction }
```

Listing 9-27

That way, an external URL like `'foo/12.xml'` will match `my_rule` and execute `mymodule/myaction` with two parameters: `$bar=12` and `$format=xml`. You can add more separators by changing the `segment_separators` parameters value in the `sfPatternRouting` factory configuration (see chapter 19).

The routing system parses the `routing.yml` file from the top to the bottom and stops at the first match. This is why you must add your own rules on top of the default ones. For instance, the URL `/foo/123` matches both of the rules defined in Listing 9-16, but symfony first tests `my_rule:`, and as that rule matches, it doesn't even test the `default:` one. The request is handled by the `mymodule/myaction` action with `bar` set to `123` (and not by the `foo/123` action).

Listing 9-16 - Rules Are Parsed Top to Bottom

```
my_rule:
  url: /foo/:bar
  param: { module: mymodule, action: myaction }

# default rules
default:
  url: /:module/:action/*
```

Listing 9-28



When a new action is created, it does not imply that you must create a routing rule for it. If the default module/action pattern suits you, then forget about the `routing.yml` file. If, however, you want to customize the action's external URL, add a new rule above the default one.

Listing 9-17 shows the process of changing the external URL format for an `article/read` action.

Listing 9-17 - Changing the External URL Format for an article/read Action

```
<?php echo url_for('article/read?id=123') ?>
=> /article/read/id/123      // Default formatting

// To change it to /article/123, add a new rule at the beginning
// of your routing.yml
article_by_id:
  url: /article/:id
  param: { module: article, action: read }
```

Listing 9-29

The problem is that the `article_by_id` rule in Listing 9-17 breaks the default routing for all the other actions of the `article` module. In fact, a URL like `article/delete` will match

this rule instead of the `default` one, and call the `read` action with `id` set to `delete` instead of the `delete` action. To get around this difficulty, you must add a pattern constraint so that the `article_by_id` rule matches only URLs where the `id` wildcard is an integer.

Pattern Constraints

When a URL can match more than one rule, you must refine the rules by adding constraints, or requirements, to the pattern. A requirement is a set of regular expressions that must be matched by the wildcards for the rule to match.

For instance, to modify the `article_by_id` rule so that it matches only URLs where the `id` parameter is an integer, add a line to the rule, as shown in Listing 9-18.

Listing 9-18 - Adding a Requirement to a Routing Rule

```
Listing 9-30 article_by_id:  
    url: /article/:id  
    param: { module: article, action: read }  
    requirements: { id: \d+ }
```

Now an `article/delete` URL can't match the `article_by_id` rule anymore, because the '`delete`' string doesn't satisfy the requirements. Therefore, the routing system will keep on looking for a match in the following rules and finally find the `default` rule.

Permalinks

A good security guideline for routing is to hide primary keys and replace them with significant strings as much as possible. What if you wanted to give access to articles from their title rather than from their ID? It would make external URLs look like this:

```
http://www.example.com/article/Finance_in_France
```

*Listing
9-31*

To that extent, you need to create a new `permalink` action, which will use a `slug` parameter instead of an `id` one, and add a new rule for it:

```
article_by_id:
    url:          /article/:id
    param:        { module: article, action: read }
    requirements: { id: \d+ }

article_by_slug:
    url:          /article/:slug
    param:        { module: article, action: permalink }
```

*Listing
9-32*

The `permalink` action needs to determine the requested article from its title, so your model must provide an appropriate method.

```
public function executePermalink($request)
{
    $article = ArticlePeer::retrieveBySlug($request->getParameter('slug'));
    $this->forward404Unless($article); // Display 404 if no article
    matches slug
    $this->article = $article;           // Pass the object to the template
}
```

*Listing
9-33*

You also need to replace the links to the `read` action in your templates with links to the `permalink` one, to enable correct formatting of internal URIs.

```
// Replace
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>

// With
<?php echo link_to('my article', 'article/
permalink?slug='.$article->getSlug()) ?>
```

*Listing
9-34*

Thanks to the `requirements` line, an external URL like `/article/Finance_in_France` matches the `article_by_slug` rule, even though the `article_by_id` rule appears first.

Note that as articles will be retrieved by slug, you should add an index to the `slug` column in the `Article` model description to optimize database performance.

Setting Default Values

You can give named wildcards a default value to make a rule work, even if the parameter is not defined. Set default values in the `param:` array.

For instance, the `article_by_id` rule doesn't match if the `id` parameter is not set. You can force it, as shown in Listing 9-19.

Listing 9-19 - Setting a Default Value for a Wildcard

*Listing
9-35*

```
article_by_id:
  url:      /article/:id
  param:    { module: article, action: read, id: 1 }
```

The default parameters don't need to be wildcards found in the pattern. In Listing 9-20, the `display` parameter takes the value `true`, even if it is not present in the URL.

Listing 9-20 - Setting a Default Value for a Request Parameter

Listing 9-36

```
article_by_id:
  url:      /article/:id
  param:    { module: article, action: read, id: 1, display: true }
```

If you look carefully, you can see that `article` and `read` are also default values for `module` and `action` variables not found in the pattern.



You can define a default parameter for all the routing rules by calling the `sfRouting::setDefaultParameter()` method. For instance, if you want all the rules to have a `theme` parameter set to `default` by default, add `$this->context->getRouting()->setDefaultParameter('theme', 'default');` to one of your global filters.

Speeding Up Routing by Using the Rule Name

The link helpers accept a rule label instead of a module/action pair if the rule label is preceded by an 'at' sign (@), as shown in Listing 9-21.

Listing 9-21 - Using the Rule Label Instead of the Module/Action

Listing 9-37

```
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>

// can also be written as
<?php echo link_to('my article', '@article_by_id?id='.$article->getId()) ?>
// or the fastest one (does not need extra parsing):
<?php echo link_to('my article', 'article_by_id', array('id' =>
$article->getId())) ?>
```

There are pros and cons to this trick. The advantages are as follows:

- The formatting of internal URIs is done 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 rule labels instead of module/action pairs.
- Using the rule label helps to abstract the logic behind an action. If you decide to change an action name but keep the URL, a simple change in the `routing.yml` file will suffice. All of the `link_to()` calls will still work without further change.
- 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_slug` than `article/display`.
- You know exactly which actions are enabled by reading the `routing.yml` file

On the other hand, a disadvantage is that adding new hyperlinks becomes less self-evident, since you always need to refer to the `routing.yml` file to find out which label is to be used for an action. And in a big project, you will certainly end with a lot of routings rules, making a bit hard to maintain them. In this case, you should package your application in several plugins, each one defining a limited set of features.

However, the experience states that using routing rules is the best choice in the long run.



During your tests (in the `dev` environment), if you want to check which rule was matched for a given request in your browser, develop the “logs” section of the web debug toolbar and look for a line specifying “matched route XXX”. You will find more information about the web debug mode in Chapter 16.

Creating Rules Without `routing.yml`

As is true of most of the configuration files, the `routing.yml` is a solution to define routing rules, but not the only one. You can define rules in PHP, but before the call to `dispatch()`, because this method determines the action to execute according to the present routing rules. Defining rules in PHP authorizes you to create dynamic rules, depending on configuration or other parameters.

The object that handles the routing rules is the `sfPatternRouting` factory. It is available from every part of the code by requiring `sfContext::getInstance()->getRouting()`. Its `prependRoute()` method adds a new rule on top of the existing ones defined in `routing.yml`. It expects two parameters: a route name and a `sfRoute` object. For instance, the `routing.yml` rule definition shown in Listing 9-18 is equivalent to the PHP code shown in Listing 9-22.

Listing 9-22 - Defining a Rule in PHP

```
sfContext::getInstance()->getRouting()->prependRoute(  
    'article_by_id', // Route name  
    new sfRoute('/article/:id', array('module' => 'article', 'action' =>  
        'read'), array('id' => '\d+')), // Route object  
);
```

Listing 9-38

The `sfRoute` class constructor takes three arguments: a pattern, an associative array of default values, and another associative array for requirements.

The `sfPatternRouting` class has other useful methods for handling routes by hand: `clearRoutes()`, `hasRoutes()` and so on. Refer to the API documentation³⁹ to learn more.



Once you start to fully understand the concepts presented in this book, you can increase your understanding of the framework by browsing the online API documentation or, even better, the symfony source. Not all the tweaks and parameters of symfony can be described in this book. The online documentation, however, is limitless.



The routing class is configurable in the `factories.yml` configuration file (to change the default routing class, see chapter 17). This chapter talks about the `sfPatternRouting` class, which is the routing class configured by default.

Dealing with Routes in Actions

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 `sfPatternRouting` object. The URIs returned by the `getCurrentInternalUri()` method can be used in a call to a `link_to()` helper, as shown in Listing 9-23.

Listing 9-23 - Using sfRouting to Get Information About the Current Route

39. http://www.symfony-project.org/api/1_4/

Listing 9-39

```
// If you require a URL like
// http://myapp.example.com/article/21

$routing = $this->getContext()->getRouting();

// Use the following in article/read action
$uri = $routing->getCurrentInternalUri();
=> article/read?id=21

$uri = $routing->getCurrentInternalUri(true);
=> @article_by_id?id=21

$rule = $routing->getCurrentRouteName();
=> article_by_id

// If you just need the current module/action names,
// remember that they are actual request parameters
$module = $request->getParameter('module');
$action = $request->getParameter('action');
```

If you need to transform an internal URI into an external URL in an action—just as `url_for()` does in a template—use the `genUrl()` method of the `sfController` object, as shown in Listing 9-24.

Listing 9-24 - Using sfController to Transform an Internal URI

Listing 9-40

```
$uri = 'article/read?id=21';

$url = $this->getController()->genUrl($uri);
=> /article/21

$url = $this->getController()->genUrl($uri, true);
=> http://myapp.example.com/article/21
```

Summary

Routing is a two-way mechanism designed to allow formatting of external URLs so that they are more user-friendly. URL rewriting is required to allow the omission of the front controller name in the URLs of one of the applications of each project. You must use link helpers each time you need to output a URL in a template if you want the routing system to work both ways. The `routing.yml` file configures the rules of the routing system and uses an order of precedence and rule requirements. The `settings.yml` file contains additional settings concerning the presence of the front controller name and a possible suffix in external URLs.

Chapter 10

Forms

Dealing with the display of form inputs, the validation of a form submission, and all the particular cases of forms is one of the most complex tasks in web development. Luckily, symfony provides a simple interface to a very powerful form sub-framework, and helps you to design and handle forms of any level of complexity in just a few lines of code.

Displaying a Form

A simple contact form featuring a name, an email, a subject and a message fields typically renders as follows:

Name	<input type="text"/>
Email	<input type="text"/>
Subject	<input style="border: none; padding: 0; margin: 0;" type="button" value="Subject A"/> <div style="border: 1px solid #ccc; padding: 5px; width: 150px; position: absolute; left: -10px; top: 0;"> Subject A Subject B Subject C </div>
Message	<input type="text"/>
<input type="button" value="Submit Query"/>	

In symfony, a form is an object defined in the action and passed to the template. In order to display a form, you must first define the fields it contains - symfony uses the term "widget". The simplest way to do it is to create a new `sfForm` object in the action method.

```
// in modules/foo/actions/actions.class.php
public function executeContact($request)
{
  $this->form = new sfForm();
  $this->form->setWidgets(array(
    'name'      => new sfWidgetFormInputText(),
    'email'     => new sfWidgetFormEmail(),
    'subject'   => new sfWidgetFormSelect(
      array(
        'label'  => 'Subject',
        'options' => array(
          'A' => 'Subject A',
          'B' => 'Subject B',
          'C' => 'Subject C'
        )
      )
    ),
    'message'   => new sfWidgetFormTextarea()
  ));
}
```

Listing 10-1

```

'email' => new sfWidgetFormInputText(array('default' =>
'me@example.com')),
'subject' => new sfWidgetFormChoice(array('choices' => array('Subject
A', 'Subject B', 'Subject C'))),
'message' => new sfWidgetFormTextarea(),
));
}
}

```

`sfForm::setWidgets()` expects an associative array of widget name / widget object. `sfWidgetFormInputText`, `sfWidgetFormChoice`, and `sfWidgetFormTextarea` are some of the numerous widget classes offered by symfony; you will find a complete list further in this chapter.

The previous example shows two widget options you can use: `default` sets the widget value, and is available for all widgets. `choices` is an option specific to the `choice` widget (which renders as a drop-down list): it defines the available options the user can select.

So the `foo/contact` action defines a form object, and then handles it to the `contactSuccess` template in a `$form` variable. The template can use this object to render the various parts of the form in HTML. The simplest way to do it is to call `echo $form`, and this will render all the fields as form controls with labels. You can also use the form object to generate the form tag:

Listing 10-2

```
// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>


|                         |  |
|-------------------------|--|
| <input type="submit" /> |  |
|-------------------------|--|


</form>
```

With the parameters passed to `setWidgets()`, symfony has enough information to display the form correctly. The resulting HTML renders exactly like the screenshot above, with this underlying code:

Listing 10-3

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


| <label for="name">Name</label></th>       | <input type="text" name="name" id="name" /></td>                                                                                                              |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <label for="email">Email</label></th>     | <input type="text" name="email" id="email" value="me@example.com" /></td>                                                                                     |
| <label for="subject">Subject</label></th> | <select name="subject" id="subject"> <option value="0">Subject A</option> <option value="1">Subject B</option> <option value="2">Subject C</option> </select> |


</form>
```

```

</tr>
<tr>
    <th><label for="message">Message</label></th>
    <td><textarea rows="4" cols="30" name="message"
id="message"></textarea></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" />
    </td>
</tr>
</table>
</form>

```

Each widget results in a table row containing a `<label>` tag, and a form input tag. Symfony deduces the label name from the widget name by uppercasing it (the `subject` widget name gives the ‘Subject’ label). As for the input tag, it depends on the widget type. Symfony adds an `id` attribute to each widget, based on its name. Lastly, the rendering of the form is always XHTML-compliant.

Customizing the Form Display

Using `echo $form` is great for prototyping, but you probably want to control exactly the resulting HTML code. The form object contains an array of fields, and calling `echo $form` actually iterates over the fields and renders them one by one. To get more control, you can iterate over the fields manually, and call the `renderRow()` on each field. The following listing produces exactly the same HTML code as previously, but the template echoes each field individually:

```

// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>


```

Listing 10-4

Rendering fields one by one allows you to change the order in which they are displayed, and also to customize their appearance. `renderRow()` expects a list of HTML attributes as first argument, so you can define a custom `class`, `id`, or JavaScript event handler for instance. The second argument of `renderRow()` is an optional label that overrides the one deduced from the widget name. Here is an example of customization for the contact form:

```

// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>


```

Listing 10-5

```
"";'), 'Your Email') ?>
<?php echo $form['message']->renderRow() ?>
<tr>
    <td colspan="2">
        <input type="submit" />
    </td>
</tr>
</table>
</form>
```

But maybe you need to output the label and the input of each field in `` tags rather than in a `<tr>` tags. A field “row” is made of a label, an optional error message (added by the validation system, explained later in this chapter), a help text, and a widget (note that the widget can consist of more than one form control). Just as you can output the various fields of a form one by one, you can also render the various parts of a form independently. Instead of using `renderRow()`, use any of `render()` (for the widget), `renderError()`, `renderLabel()`, and `renderHelp()`. For instance, if you want to render the whole form with `` tags, write the template as follows:

Listing 10-6

```
// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>
<ul>
    <?php foreach ($form as $field): ?>
    <li>
        <?php echo $field->renderLabel() ?>
        <?php echo $field->render() ?>
    </li>
    <?php endforeach; ?>
    <li>
        <input type="submit" />
    </li>
</ul>
</form>
```

This renders to HTML as follows:

Listing 10-7

```
<form action="/frontend_dev.php/foo/contact" method="POST">
<ul>
    <li>
        <label for="name">Name</label>
        <input type="text" name="name" id="name" />
    </li>
    <li>
        <label for="email">Email</label>
        <input type="text" name="email" id="email" />
    </li>
    <li>
        <label for="subject">Subject</label>
        <select name="subject" id="subject">
            <option value="0">Subject A</option>
            <option value="1">Subject B</option>
            <option value="2">Subject C</option>
        </select>
    </li>
    <li>
        <label for="message">Message</label>
        <textarea rows="4" cols="30" name="message" id="message"></textarea>
    </li>
```

```
<li>
    <input type="submit" />
</li>
</ul>
</form>
```



A field row is the representation of all the elements of a form field (label, error message, help text, form input) by a formatter. By default, symfony uses a `table` formatter, and that's why `renderRow()` returns a set of `<tr>`, `<th>` and `<td>` tags. Alternatively, you can obtain the same HTML code as above by simply specifying an alternative `list` formatter for the form, as follows:

```
// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>
<ul>
    <?php echo $form->renderUsing('list') ?>
    <li>
        <input type="submit" />
    </li>
</ul>
</form>
```

Listing 10-8



Check the API documentation of the `sfWidgetFormSchemaFormatter` class to learn how to create your own formatter.

Form Widgets

There are many available form widgets at your disposal to compose your forms. All widgets accept at least the `default` option.

You can also define the label of a widget, and even its HTML attributes, when you create the form:

```
$this->form = new sfForm();
$this->form->setWidgets(array(
    'name'      => new sfWidgetFormInput(array('label' => 'Your Name'),
array('size' => 25, 'class' => 'foo')),
    'email'     => new sfWidgetFormInput(array('default' => 'me@example.com',
'label' => 'Your Email'), array('onclick' => 'this.value = "";"')),
    'subject'   => new sfWidgetFormChoice(array('choices' => array('Subject
A', 'Subject B', 'Subject C'))),
    'message'   => new sfWidgetFormTextarea(array(), array('rows' => '20',
'cols' => 5)),
));
```

Listing 10-9

Symfony uses these parameters to display the widget, and you can still override them by passing custom parameters to `renderRow()` in the template.



As an alternative to calling `setWidgets()` with an associative array, you can call the `setWidget($name, $widget)` method several times.

Standard Widgets

Here is a list of available widget types, and how they translate into HTML (via `renderRow()`):

```
Listing 10-10
// Text input
$form->setWidget('full_name', new sfWidgetFormInput(array('default' => 'John Doe')));
    <label for="full_name">Full Name</label>
    <input type="text" name="full_name" id="full_name" value="John Doe" />

// Textarea
$form->setWidget('address', new sfWidgetFormTextarea(array('default' => 'Enter your address here'), array('cols' => 20, 'rows' => 5)));
    <label for="address">Address</label>
    <textarea name="address" id="address" cols="20" rows="5">Enter your address here</textarea>

// Password input
// Note that 'password' type widgets don't take a 'default' parameter for security reasons
$form->setWidget('pwd', new sfWidgetFormInputPassword());
    <label for="pwd">Pwd</label>
    <input type="password" name="pwd" id="pwd" />

// Hidden input
$form->setWidget('id', new sfWidgetFormInputHidden(array('default' => 1234)));
    <input type="hidden" name="id" id="id" value="1234" />

// Checkbox
$form->setWidget('single', new sfWidgetFormInputCheckbox(array('value_attribute_value' => 'single', 'default' => true)));
    <label for="single">Single</label>
    <input type="checkbox" name="single" id="single" value="true" checked="checked" />
```

There are more options available for each widget than what is exposed here. Check the widget API documentation for a complete description of what each widget expects and how it renders.

List Widgets

Whenever users have to make a choice in a list of values, and whether they can select one or many option in this list, a single widget answers all the needs: the `choice` widget. According to two optional parameters (`multiple` and `expanded`), this widget renders in a different way:

	<code>multiple=false (default)</code>	<code>multiple=true</code>
<code>expanded=false (default)</code>	Dropdown list (`<select>`)	Dropdown box (`<select multiple>`)
<code>expanded=true</code>	List of Radiobuttons	List of checkboxes

The `choice` widget expects at least a `choices` parameter with an associative array to define both the value and the text of each option. Here is an example of each syntax:

```
// Dropdown list (select)
$form->setWidget('country', new sfWidgetFormChoice(array(
    'choices'  => array('' => 'Select from the list', 'us' => 'USA', 'ca'
=> 'Canada', 'uk' => 'UK', 'other'),
    'default'   => 'uk'
)));
// symfony renders the widget in HTML as
<label for="country">Country</label>
<select id="country" name="country">
    <option value="">Select from the list</option>
    <option value="us">USA</option>
    <option value="ca">Canada</option>
    <option value="uk" selected="selected">UK</option>
    <option value="0">other</option>
</select>

// Dropdown list with multiple choices
$form->setWidget('languages', new sfWidgetFormChoice(array(
    'multiple' => 'true',
    'choices'  => array('en' => 'English', 'fr' => 'French', 'other'),
    'default'   => array('en', 0)
));
// symfony renders the widget in HTML as
<label for="languages">Language</label>
<select id="languages" multiple="multiple" name="languages[]">
    <option value="en" selected="selected">English</option>
    <option value="fr">French</option>
    <option value="0" selected="selected">other</option>
</select>

// List of Radio buttons
$form->setWidget('gender', new sfWidgetFormChoice(array(
    'expanded' => true,
    'choices'  => array('m' => 'Male', 'f' => 'Female'),
    'class'     => 'gender_list'
)));
// symfony renders the widget in HTML as
<label for="gender">Gender</label>
<ul class="gender_list">
    <li><input type="radio" name="gender" id="gender_m" value="m"><label
for="gender_m">Male</label></li>
    <li><input type="radio" name="gender" id="gender_f" value="f"><label
for="gender_f">Female</label></li>
</ul>

// List of checkboxes
$form->setWidget('interests', new sfWidgetFormChoice(array(
    'multiple' => 'true',
    'expanded' => true,
    'choices'  => array('Programming', 'Other')
)));
// symfony renders the widget in HTML as
<label for="interests">Interests</label>
<ul class="interests_list">
    <li><input type="checkbox" name="interests[]" id="interests_0"
    &gt;
```

*Listing
10-12*

```
value="0">><label for="interests_0">Programming</label></li>
<li><input type="checkbox" name="interests[]" id="interests_1"
value="1">><label for="interests_1">Other</label></li>
</ul>
```



You probably noticed that symfony automatically defines an `id` attribute for each form input, based on a combination of the name and value of the widget. You can override the `id` attribute widget by widget, or alternatively set a global rule for the whole form with the `setIdFormat()` method of the `sfWidgetForm` class:

Listing 10-13

```
// in modules/foo/actions/actions.class.php
$this->form = new sfForm();
$this->form->getWidgetSchema()->setIdFormat('my_form_%s');
```

Foreign Key Widgets

When editing Model objects through a form, a particular list of choices always comes up: the list of objects that can be related to the current one. This happens when models are related by a many-to-one relationship, or a many-to-many. Fortunately, the `sfPropelPlugin` bundled with symfony provides a `sfWidgetFormPropelChoice` widget specifically for these cases (and `sfDoctrinePlugin` offers a similar `sfWidgetFormDoctrineChoice` widget).

For instance, if a `Section` has many `Articles`, you should be able to choose a section among the list of existing ones when editing an article. To do so, an `ArticleForm` should use the `sfWidgetFormPropelChoice` widget:

Listing 10-14

```
$articleForm = new sfForm();
$articleForm->setWidgets(array(
    'id'      => sfWidgetFormInputHidden(),
    'title'   => sfWidgetFormInputText(),
    'section_id' => sfWidgetFormPropelChoice(array(
        'model'  => 'Section',
        'column' => 'name'
    ))
));
```

This will display a list of existing sections... provided you defined a `__toString()` method in the `Section` model class. That's because symfony first retrieves the available `Section` objects, and populates a `choice` widget with them by trying to echo each object. So the `Section` model should at least feature the following method:

Listing 10-15

```
// in lib/model/Section.php
public function __toString()
{
    return $this->getName();
}
```

The `sfWidgetFormPropelChoice` widget is an extension of the `sfWidgetFormChoice` widget, so you can use the `multiple` option to deal with many-to-many relationships, and the `expanded` option to change the way the widget is rendered.

If you want to order the list of choices in a special way, or filter it so that it displays only a portion of the available choices, use the `criteria` option to pass a `Criteria` object to the widget. Doctrine supports the same kind of customization: you can pass a `Doctrine_Query` object to the widget with the `query` option.

Date Widgets

Date and time widgets output a set of drop-down lists, populated with the available values for the day, month, year, hour or minute.

```
// Date
$years = range(1950, 1990);
$form->setWidget('dob', new sfWidgetFormDate(array(
    'label'    => 'Date of birth',
    'default'  => '01/01/1950', // can be a timestamp or a string
understandable by strtotime()
    'years'    => array_combine($years, $years)
)));
// symfony renders the widget in HTML as
<label for="dob">Date of birth</label>
<select id="dob_month" name="dob[month]">
    <option value="" />
    <option selected="selected" value="1">01</option>
    <option value="2">02</option>
    ...
    <option value="12">12</option>
</select> /
<select id="dob_day" name="dob[day]">
    <option value="" />
    <option selected="selected" value="1">01</option>
    <option value="2">02</option>
    ...
    <option value="31">31</option>
</select> /
<select id="dob_year" name="dob[year]">
    <option value="" />
    <option selected="selected" value="1950">1950</option>
    <option value="1951">1951</option>
    ...
    <option value="1990">1990</option>
</select>

// Time
$form->setWidget('start', new sfWidgetFormTime(array('default' =>
'12:00')));
// symfony renders the widget in HTML as
<label for="start">Start</label>
<select id="start_hour" name="start[hour]">
    <option value="" />
    <option value="0">00</option>
    ...
    <option selected="selected" value="12">12</option>
    ...
    <option value="23">23</option>
</select> :
<select id="start_minute" name="start[minute]">
    <option value="" />
    <option selected="selected" value="0">00</option>
    <option value="1">01</option>
    ...
    <option value="59">59</option>
</select>
```

*Listing
10-16*

```
// Date and time
$form->setWidget('end', new sfWidgetFormDateTime(array('default' => '01/01/2008 12:00')));
// symfony renders the widget in HTML as 5 dropdown lists for month, day, year, hour, minute
```

Of course, you can customize the date format, to display it in European style instead of International style (%day%/%month%/%year% instead of %month%/%day%/%year%), you can switch to 2x12 hours per day instead of 24 hours, you can define custom values for the first option of each dropdown box, and you can define limits to the possible values. Once again, check the API documentation for more details about the options of the date and time widgets.

Date widgets are a good example of the power of widgets in symfony. A widget is not a simple form input. It can be a combination of several inputs, that symfony can render and read from in a transparent way.

I18n Widgets

In multilingual applications, dates must be displayed in a format according to the user culture (see Chapter 13 for details about culture and localization). To facilitate this localization in forms, symfony provides an `sfWidgetFormI18nDate` widget, which expects a user culture to decide of the date formatting parameters. You can also specify a `month_format` to have the month drop-down display month names (in the user language) instead of numbers.

Listing 10-17

```
// Date
$years = range(1950, 1990);
$form->setWidget('dob', new sfWidgetFormI18nDate(array(
    'culture'      => $this->getUser()->getCulture(),
    'month_format' => 'name', // Use any of 'name' (default),
    'short_name', and 'number'
    'label'         => 'Date of birth',
    'default'       => '01/01/1950',
    'years'         => array_combine($years, $years)
));
// For an English-speaking user, symfony renders the widget in HTML as
<label for="dob">Date of birth</label>
<select id="dob_month" name="dob[month]">
    <option value="" />
    <option selected="selected" value="1">January</option>
    <option value="2">February</option>
    ...
    <option value="12">December</option>
</select> /
<select id="dob_day" name="dob[day]">...</select> /
<select id="dob_year" name="dob[year]">...</select>
// For an French-speaking user, symfony renders the widget in HTML as
<label for="dob">Date of birth</label>
<select id="dob_day" name="dob[day]">...</select> /
<select id="dob_month" name="dob[month]">
    <option value="" />
    <option selected="selected" value="1">Janvier</option>
    <option value="2">Février</option>
    ...
    <option value="12">Décembre</option>
</select> /
<select id="dob_year" name="dob[year]">...</select>
```

Similar widgets exist for time (`sfWidgetFormI18nTime`), and datetime (`sfWidgetFormI18nDateTime`).

There are two drop-down lists that occur in many forms and that also rely on the user culture: country and language selectors. Symfony provides two widgets especially for this purpose. You don't need to define the choices in these widgets, as symfony will populate them with the list of countries and languages in the language of the user (provided the user speaks any of the 250 referenced languages in symfony).

```
// Country list
$form->setWidget('country', new
sfWidgetFormI18nCountryChoice(array('default' => 'UK')));
// For an English-speaking user, symfony renders the widget in HTML as
<label for="country">Country</label>
<select id="country" name="country">
  <option value="" />
  <option value="AD">Andorra</option>
  <option value="AE">United Arab Emirates</option>
  ...
  <option value="ZWD">Zimbabwe</option>
</select>

// Language list
$form->setWidget('language', new sfWidgetFormI18nLanguageChoice(array(
  'languages' => array('en', 'fr', 'de'), // optional restricted list of
  languages
  'default'    => 'en'
)));
// For an English-speaking user, symfony renders the widget in HTML as
<label for="language">Language</label>
<select id="language" name="language">
  <option value="" />
  <option value="de">German</option>
  <option value="en" selected="selected">English</option>
  <option value="fr">French</option>
</select>
```

*Listing
10-18*

File Widgets

Dealing with file input tags is not more complicated than dealing with other widgets:

```
// Input file
$form->setWidget('picture', new sfWidgetFormInputFile());
// symfony renders the widget in HTML as
<label for="picture">Picture</label>
<input id="picture" type="file" name="picture"/>
// Whenever a form has a file widget, renderFormTag() outputs a <form> tag
with the multipart option

// Editable input file
$form->setWidget('picture', new
sfWidgetFormInputFileEditable(array('default' => '/images/foo.png')));
// symfony renders the widget in HTML as a file input tag, together with a
preview of the current file
```

*Listing
10-19*



Third-party plugins provide many additional widgets. You can easily find a rich text editor widget, a calendar widget, or other “rich UI” widgets for various JavaScript libraries. Check the Plugins repository⁴⁰ for more details.

Handling a Form Submission

When users fill a form and submit it, the web application server needs to retrieve the data from the request and do some stuff with it. The `sfForm` class provides all the necessary methods to do that in a couple lines of code.

Simple Form Handling

Since widgets output as regular HTML form fields, getting their value in the action that handles the form submission is as easy as checking the related request parameters. For the example contact form, the action could look like this:

Listing 10-20

```
// in modules/foo/actions/actions.class.php
public function executeContact($request)
{
    // Define the form
    $this->form = new sfForm();
    $this->form->setWidgets(array(
        'name' => new sfWidgetFormInputText(),
        'email' => new sfWidgetFormInput(array('default' =>
'me@example.com')),
        'subject' => new sfWidgetFormChoice(array('choices' => array('Subject
A', 'Subject B', 'Subject C'))),
        'message' => new sfWidgetFormTextarea(),
    ));

    // Deal with the request
    if ($request->isMethod('post'))
    {
        // Handle the form submission
        $name = $request->getParameter('name');
        // Do stuff
        // ...
        $this->redirect('foo/bar');
    }
}
```

If the request method is ‘GET’, this action will terminate over a `sfView::SUCCESS` and therefore render the `contactSuccess` template to display the form. If the request method is ‘POST’, then the action handles the form submission and redirects to another action. For this to work, the `<form>` target action must be the same as the one displaying it. That explains why the previous examples used `foo/contact` as a form target:

Listing 10-21

```
// in modules/foo/templates/contactSuccess.php
<?php echo $form->renderFormTag('foo/contact') ?>
...
```

40. <http://www.symfony-project.org/plugins/>

Form Handling With Data Validation

In practice, there is much more to form submission handling than just getting the values entered by the user. For most form submissions, the application controller needs to:

1. Check that the data is conform to a set of predefined rules (required fields, format of the email, etc.)
2. Optionally transform some of the input data to make it understandable (trim whitespaces, convert dates to PHP format, etc)
3. If the data is not valid, display the form again, with error messages where applicable
4. If the data is correct, do some stuff and then redirect to another action

Symfony provides an automatic way to validate the submitted data against a set of predefined rules. First, define a set of validators for each field. Second, when the form is submitted, “bind” the form object with the user submitted values (i.e., retrieve the values submitted by the user and put them in the form). Lastly, ask the form to check that the data is valid. The following example shows how to check that the value retrieved from the `email` widget is, indeed, an email address, and to check that the `message` has a minimum size of 4 characters:

```
// in modules/foo/actions/actions.class.php
public function executeContact($request)
{
    // Define the form
    $this->form = new sfForm();
    $this->form->setWidgets(array(
        'name'      => new sfWidgetFormInputText(),
        'email'     => new sfWidgetFormInput(array('default' =>
'me@example.com')),
        'subject'   => new sfWidgetFormChoice(array('choices' => array('Subject
A', 'Subject B', 'Subject C'))),
        'message'   => new sfWidgetFormTextarea(),
    ));
    $this->form->setValidators(array(
        'name'      => new sfValidatorString(),
        'email'     => new sfValidatorEmail(),
        'subject'   => new sfValidatorString(),
        'message'   => new sfValidatorString(array('min_length' => 4))
    ));

    // Deal with the request
    if ($request->isMethod('post'))
    {
        $this->form->bind(/* user submitted data */);
        if ($this->form->isValid())
        {
            // Handle the form submission
            // ...

            $this->redirect('foo/bar');
        }
    }
}
```

*Listing
10-22*

`setValidators()` uses a similar syntax to the `setWidgets()` method. `sfValidatorEmail` and `sfValidatorString` are two of the numerous symfony validator classes, listed further in this chapter. Naturally, `sfForm` also provide a `setValidator()` method to add validators one by one.

To put the request data into the form and bind them together, use the `sfForm::bind()` method. A form must be bound with some data to check its validity.

`isValid()` checks that all the registered validators pass. If this is the case, `isValid()` returns `true`, and the action can proceed with the form submission. If the form is not valid, then the action terminates with the default `sfView::SUCCESS` and displays the form again. But the form isn't just displayed with the default values, as the first time it was displayed. The form inputs show up filled with the data previously entered by the user, and error messages appear wherever the validators didn't pass.

Name	<input type="text"/>
Email	<ul style="list-style-type: none">• The email address is invalid. <input type="text" value="fabien"/>
Subject	<input type="text" value="Subject A"/>
Message	<ul style="list-style-type: none">• The message field is required. <input type="text"/>
<input type="button" value="Submit Query"/>	



The validation process doesn't stop when the form meets an invalid field. `isValid()` processes the whole form data and checks all the fields for errors, to avoid displaying new error messages as the user corrects its mistakes and submits the form again.

Using Clean Form Data

In the previous listing, we haven't defined the request data received by the form during the binding process. The problem is that the request contains more than just the form data. It also contains headers, cookies, parameters passed as GET arguments, and all this might pollute the binding process. A good practice is to pass only the form data to the `bind()` method.

Fortunately, symfony offers a way to name all form inputs using an array syntax. Define the name attribute format width the `setNameFormat()` method in the action when you define the form, as follows:

Listing 10-23

```
// in modules/foo/actions/actions.class.php
// Define the form
$this->form->getWidgetSchema()->setNameFormat('contact[%s]');
```

That way, all the generated form inputs render with a name like `form[WIDGET_NAME]` instead of just `WIDGET_NAME`:

```
<label for="contact_name">Name</label>
<input type="text" name="contact[name]" id="contact_name" />
...
<label for="contact_email">Email</label>
<input type="text" name="contact[email]" id="contact_email"
value="me@example.com" />
...
<label for="contact_subject">Subject</label>
<select name="contact[subject]" id="contact_subject">
  <option value="0">Subject A</option>
  <option value="1">Subject B</option>
  <option value="2">Subject C</option>
</select>
...
<label for="contact_message">Message</label>
<textarea rows="4" cols="30" name="contact[message]"
id="contact_message"></textarea>
```

*Listing
10-24*

The action can now retrieve the `contact` request parameter into a single variable. This variable contains an array of all the data entered by the user in the form:

```
// in modules/foo/actions/actions.class.php
// Deal with the request
if ($request->isMethod('post'))
{
    $this->form->bind($request->getParameter('contact'));
    if ($this->form->isValid())
    {
        // Handle the form submission
        $contact = $this->form->getValues();
        $name = $contact['name'];

        // Or to get a specific value
        $name = $this->form->getValue('name');

        // Do stuff
        // ...
        $this->redirect('foo/bar');
    }
}
```

*Listing
10-25*

When the `bind()` method receives an array of parameters, symfony automatically avoid injection of additional fields on the client side. This security feature will make the form validation fail if the array of `contact` parameters contains a field that is not in the original form definition.

You will notice one more difference in the action code above with the one written previously. The action uses the array of values passed by the form object (`$form->getValues()`) rather than the one from the request. This is because the validators have the ability to filter the input and clean it, so it's always better to rely on the data retrieved from the form object (by way of `getValues()` or `getValue()`) than the data from the request. And for composite fields (like date widgets), the data returned by `getValues()` is already recomposed into the original names:

```
// When submitted, the form controls of a 'date' widget...
<label for="contact_dob">Date of birth</label>
<select id="contact_dob_month" name="contact[dob][month]">...</select> /
<select id="contact_dob_day" name="contact[dob][day]">...</select> /
```

*Listing
10-26*

```
<select id="contact_dob_year" name="contact[dob][year]">...</select>
// ...result in three request parameters in the action
$contact = $request->getParameter('contact');
$month = $contact['dob']['month'];
$day = $contact['dob']['day'];
$year = $contact['dob']['year'];
$dateOfBirth = mktime(0, 0, 0, $month, $day, $year);
// But if you use getValues(), you can retrieve directly a correct date
$contact = $this->form->getValues();
$dateOfBirth = $contact['dob'];
```

So take the habit to always use an array syntax for your form fields (using `setNameFormat()`) and to always use the clean form output (using `getValues()`).

Customizing Error Messages Display

Where do the error messages shown in the screenshot above come from? You know that a widget is made of four components, and the error message is one of them. In fact, the default (table) formatter renders a field row as follows:

Listing 10-27

```
<?php if ($field->hasError()): ?>
<tr>
  <td colspan="2">
    <?php echo $field->renderError() ?>          // List of errors
  </td>
</tr>
<?php endif; ?>
<tr>
  <th><?php echo $field->renderLabel() ?></th>    // Label
  <td>
    <?php echo $field->render() ?>                // Widget
    <?php if ($field->hasHelp()): ?>
      <br /><?php echo $field->renderHelp() ?>        // Help
    <?php endif; ?>
  </td>
</tr>
```

Using any of the methods above, you can customize where and how the error messages appear for each field. In addition, you can display a global error message on top of the form if it is not valid:

Listing 10-28

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

Customizing Validators

In a form, all fields must have a validator and by default, all the fields are required. If you need to set a field optional, pass the `required` option to the validator and set it to `false`. For instance, the following listing shows how to make the `name` field required and the `email` field optional:

Listing 10-29

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

```
'message' => new sfValidatorString(array('min_length' => 4))
));
```

You can apply more than one validator on a field. For instance, you may want to check that the `email` field satisfies both the `sfValidatorEmail` and the `sfValidatorString` validators with a minimum size of 4 characters. In such a case, use the `sfValidatorAnd` validator to combine two validators, and pass the two `sfValidatorEmail` and `sfValidatorString` validators as an argument:

```
$this->form->setValidators(array(
    'name'      => new sfValidatorString(),
    'email'     => new sfValidatorAnd(array(
        new sfValidatorEmail(),
        new sfValidatorString(array('min_length' => 4)),
    ), array('required' => false)),
    'subject'   => new sfValidatorString(),
    'message'   => new sfValidatorString(array('min_length' => 4))
));
```

Listing 10-30

If both validators are valid, then the `email` field is declared valid. Similarly, you can use the `sfValidatorOr` validator to combine several validators. If one of the validators is valid, then the field is declared valid.

Each invalid validator results into an error message in the field. These error messages are in English but use the symfony internationalization helpers; if your project uses other languages, you can easily translate the error messages with an i18n dictionary. Alternatively, every validator provides a third argument to customize its error messages. Each validator has at least two error messages: the `required` message and the `invalid` message. Some validators can display error messages for a different purpose, and will always support the overriding of the error messages through their third argument:

```
// in modules/foo/actions/actions.class.php
$this->form->setValidators(array(
    'name'      => new sfValidatorString(),
    'email'     => new sfValidatorEmail(array(), array(
        'required'  => 'Please provide an email',
        'invalid'   => 'Please provide a valid email address (me@example.com)'
    )),
    'subject'   => new sfValidatorString(),
    'message'   => new sfValidatorString(array('min_length' => 4), array(
        'required'  => 'Please provide a message',
        'min_length' => 'Please provide a longer message (at least 4
characters)'
    )));
));
```

Listing 10-31

Naturally, these custom messages will render in the templates through i18n helpers, so any multilingual application can also translate custom error messages in a dictionary (see Chapter 13 for details).

Applying a Validator To Several Fields

The syntax used above to define validators on a form does not allow to validate that two fields are valid *together*. For instance, in a registration form, there are often two `password` fields that must match, otherwise the registration is refused. Each password field is not valid on its own, it is only valid when associated with the other field.

That's why you can set a 'multiple' validator via `setPostValidator()` to set the validators that work on several values. The post validator is executed after all other validators and receives an array of cleaned up values. If you need to validate raw input form data, use the `setPreValidator()` method instead.

A typical registration form definition would look like this:

```
Listing 10-32 // in modules/foo/actions/actions.class.php
// Define the form
$this->form = new sfForm();
$this->form->setWidgets(array(
    'login'      => new sfWidgetFormInputText(),
    'password1'  => new sfWidgetFormInputText(),
    'password2'  => new sfWidgetFormInputText()
));
$this->form->setValidators(array(
    'login'      => new sfValidatorString(), // login is required
    'password1'  => new sfValidatorString(), // password1 is required
    'password2'  => new sfValidatorString(), // password2 is required
));
$this->form->setPostValidators(new sfValidatorSchemaCompare('password1',
    '==', 'password2'));
```

The `sfValidatorSchemaCompare` validator is a special multiple validator that receives all the cleaned up values and can pick up two of them for comparison. Naturally, you can define more than one post validator by using the `sfValidatorAnd` and the `sfValidatorOr` validators.

Validators

Symfony offers quite a lot of validators. Remember that each validator accepts an array of option and an array of errors as arguments where you can at least customize the required and invalid error messages.

```
Listing 10-33 // String validator
$form->setValidator('message', new sfValidatorString(array(
    'min_length' => 4,
    'max_length' => 50,
),
array(
    'min_length' => 'Please post a longer message',
    'max_length' => 'Please be less verbose',
)));

// Number validator
$form->setValidator('age', new sfValidatorNumber(array( // use
    'sfValidatorInteger' instead if you want to force integer values
    'min' => 18,
    'max' => 99.99,
),
array(
    'min' => 'You must be 18 or more to use this service',
    'max' => 'Are you kidding me? People over 30 can\'t even use the
    Internet',
))));
```

```
// Email validator
$form->setValidator('email', new sfValidatorEmail());

// URL validator
$form->setValidator('website', new sfValidatorUrl());

// Regular expression validator
$form->setValidator('IP', new sfValidatorRegex(array(
    'pattern' => '^[0-9]{3}\.[0-9]{3}\.[0-9]{2}\.[0-9]{3}$'
))');
```

Even though some form controls (like drop-down lists, checkboxes, radio button groups) restrict the possible choices, a malicious user can always try to hack your forms by manipulating the page with Firebug or submitting a query with a scripting language. Consequently, you should also validate fields that only accept a limited array of values:

```
// Boolean validator
$form->setValidator('has_signed_terms_of_service', new
sfValidatorBoolean());

// Choice validator (to restrict values in a list)
$form->setValidator('subject', new sfValidatorChoice(array(
    'choices' => array('Subject A', 'Subject B', 'Subject C')
)));

// Multiple choice validator
$form->setValidator('languages', new sfValidatorChoice(array(
    'multiple' => true,
    'choices' => array('en' => 'English', 'fr' => 'French', 'other')
)));
```

*Listing
10-34*

I18n choice validators exist for country lists (`sfValidatorI18nChoiceCountry`) and language lists (`sfValidatorI18nChoiceLanguage`). These validators accept a restricted list of countries and languages if you want to limit the possible options.

The `sfValidatorChoice` validator is often used to validate a `sfWidgetFormChoice` widget. And since you can use the `sfWidgetFormChoice` widget for foreign key columns, so symfony also provides a validator to check that the foreign key value exists in the foreign table:

```
// Propel choice validator
$form->setValidator('section_id', new sfValidatorPropelChoice(array(
    'model' => 'Section',
    'column' => 'name'
)));

// Doctrine choice validator
$form->setValidator('section_id', new sfValidatorDoctrineChoice(array(
    'model' => 'Section',
    'column' => 'name'
)));
```

*Listing
10-35*

Another useful Model-related validator is the `sfValidatorPropelUnique` validator, which checks that a new value entered via a form doesn't conflict with an existing value in a database column with a unique index. For instance, two users cannot have the same login, so when editing a `User` object with a form, you must add a `sfValidatorPropelUnique` validator on this column:

Listing 10-36

```
// Propel unique validator
$form->setValidator('nickname', new sfValidatorPropelUnique(array(
    'model' => 'User',
    'column' => 'login'
)));

$form->setValidator('nickname', new sfValidatorDoctrineUnique(array(
    'model' => 'User',
    'column' => 'login'
)));
```

To make your forms even more secure and avoid Cross-Site Request Forgery⁴¹ attacks, you can enable the CSRF protection:

Listing 10-37

```
// CSRF protection - set the secret to a random string that nobody knows
$form->addCSRFProtection('flkd445rvvrGV34G');
```



You can set the CSRF secret once for the whole site in the `settings.yml` file:

Listing 10-38

```
# in apps/myapp/config/settings.yml
all:
    .settings:
        # Form security secret (CSRF protection)
        csrf_secret:      ##CSRF_SECRET##      # Unique secret to enable CSRF
protection or false to disable
```

The multiple validators work with the whole form, rather than a single input. Here is a list of available multiple validators:

Listing 10-39

```
// compare validator - compare two fields
$form->setPostValidator(new sfValidatorSchemaCompare('password1', '==',
'password2'));

// Extra field validator: looks for fields in the request not present in
the form
$form->setOption('allow_extra_fields', false);
$form->setOption('filter_extra_fields', true);
```

Alternative Ways to Use a Form

Form Classes

With all the widget options, validators and form parameters, the contact form definition written in the actions class looks quite messy:

Listing 10-40

```
// in modules/foo/actions/actions.class.php
// Define the form
$this->form = new sfForm();
$this->form->getWidgetSchema()->setNameFormat('contact[%s]');
$this->form->getWidgetSchema()->setIdFormat('my_form_%s');
```

41. [http://en.wikipedia.org/wiki/Cross-site request forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery)

```
$this->form->setWidgets(array(
    'name'      => new sfWidgetFormInputText(),
    'email'     => new sfWidgetFormInput(array('default' => 'me@example.com'))),
    'subject'   => new sfWidgetFormChoice(array('choices' => array('Subject
A', 'Subject B', 'Subject C'))),
    'message'   => new sfWidgetFormTextarea(),
));
$this->form->setValidators(array(
    'name'      => new sfValidatorString(),
    'email'     => new sfValidatorEmail(),
    'subject'   => new sfValidatorString(),
    'message'   => new sfValidatorString(array('min_length' => 4))
));
```

The best practice is to create a form class with the same properties and instantiate it in all the actions using it. For instance, here is how you could create a class for the contact form:

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

    public function configure()
    {
        $this->widgetSchema->setNameFormat('contact[%s]');
        $this->widgetSchema->setIdFormat('my_form_%s');
        $this->setWidgets(array(
            'name'      => new sfWidgetFormInputText(),
            'email'     => new sfWidgetFormInput(array('default' =>
'me@example.com'))),
            'subject'   => new sfWidgetFormChoice(array('choices' =>
array('Subject A', 'Subject B', 'Subject C'))),
            'message'   => new sfWidgetFormTextarea(),
));
        $this->setValidators(array(
            'name'      => new sfValidatorString(),
            'email'     => new sfValidatorEmail(),
            'subject'   => new sfValidatorString(),
            'message'   => new sfValidatorString(array('min_length' => 4))
));
        $this->setDefaults(array(
            'email' => 'me@example.com'
        ));
    }
}
```

*Listing
10-41*

Now getting a contact form object in the action has never been easier:

```
// in modules/foo/actions/actions.class.php
// Define the form
$this->form = new ContactForm();
```

*Listing
10-42*

Altering a Form Object

When you use a form class definition, the form is defined outside the action. That makes dynamic default value assignment quite difficult. That's why the form object takes an array of default values as its first argument:

```
Listing 10-43 // in modules/foo/actions/actions.class.php
// Define the form
$this->form = new ContactForm(array('email' => 'me@example.com'));
```

You can also override existing widget or validator settings by calling `setWidget()` or `setValidator()` on an existing field name.

However, widgets and validators are objects in symfony, and offer a clean API to modify their properties:

```
Listing 10-44 // in modules/foo/actions/actions.class.php
// Define the form
$this->form = new ContactForm();

// Allow multiple language selections
$form->getWidget('language')->setOption('multiple', true);
// Add a 'gender' list of options widget
$form->setWidget('gender', new sfWidgetFormChoice(array('expanded' => true, 'choices' => array('m' => 'Male', 'f' => 'Female')), array('class' => 'gender_list')));
// Change the HTML attributes of the 'subject' widget
$form->getWidget('subject')->setAttribute('disabled', 'disabled');
// Remove the 'subject' field
unset($form['subject'])
// Note: You cannot remove just the widget. Removing a widget will also
remove the related validators

// Change the 'min_length' error in the 'message' validator
$form->getValidator('message')->setMessage('min_length', 'Message too
short');
// Make the 'name' field optional
$form->getValidator('name')->setOption('required', false);
```

Custom Widget and Validator classes

A custom widget is simply a class extending `sfWidgetForm`, and providing a `configure()` and a `render()` methods. Check the code of existing widget classes for a deeper understanding of the widgets system. The next listing exposes the code of the `sfWidgetFormInput` widget to illustrate the widget structure:

```
Listing 10-45 class sfWidgetFormInputText extends sfWidgetForm
{
    /**
     * Configures the current widget.
     * This method allows each widget to add options or HTML attributes
     * during widget creation.
     * Available options:
     *   * type: The widget type (text by default)
     *
     *   * @param array $options      An array of options
     *   * @param array $attributes  An array of default HTML attributes
```

```

 * @see sfWidgetForm
 */
protected function configure($options = array(), $attributes = array())
{
    $this->addOption('type', 'text');
    $this->setOption('is_hidden', false);
}

/**
 * Renders the widget as HTML
 *
 * @param string $name      The element name
 * @param string $value     The value displayed in this widget
 * @param array  $attributes An array of HTML attributes to be merged
 * with the default HTML attributes
 * @param array  $errors    An array of errors for the field
 * @return string An HTML tag string
 * @see sfWidgetForm
 */
public function render($name, $value = null, $attributes = array(),
$errors = array())
{
    return $this->renderTag('input', array_merge(
        array('type' => $this->getOption('type'), 'name' => $name, 'value'
=> $value),
        $attributes
    ));
}
}

```

A validator class must extend `sfValidatorBase` and provide a `configure()` and a `doClean()` methods. Why `doClean()` and not `validate()`? Because validators do two things: they check that the input fulfills a set of rules, and they optionally clean the input (for instance by forcing the type, trimming, converting date strings to timestamp, etc.). So the `doClean()` method must return the cleaned input, or throw a `sfValidatorError` exception if the input doesn't satisfy any of the validator rules. Here is an illustration of this concept, with the code of the `sfValidatorInteger` validator.

```

class sfValidatorInteger extends sfValidatorBase
{
    /**
     * Configures the current validator.
     * This method allows each validator to add options and error messages
     * during validator creation.
     * Available options:
     * * max: The maximum value allowed
     * * min: The minimum value allowed
     * Available error codes:
     * * max
     * * min
     *
     * @param array $options  An array of options
     * @param array $messages An array of error messages
     * @see sfValidatorBase
     */
protected function configure($options = array(), $messages = array())
{
    $this->addOption('min');
}

```

*Listing
10-46*

```
$this->addOption('max');
$this->addMessage('max', '"%value%" must be less than %max%.');
$this->addMessage('min', '"%value%" must be greater than %min%.');
$this->setMessage('invalid', '"%value%" is not an integer.');
}

/**
 * Cleans the input value.
 *
 * @param mixed $value The input value
 * @return mixed The cleaned value
 * @throws sfValidatorError
 */
protected function doClean($value)
{
    $clean = intval($value);
    if (strval($clean) != $value)
    {
        throw new sfValidatorError($this, 'invalid', array('value' =>
$value));
    }
    if ($this->hasOption('max') && $clean > $this->getOption('max'))
    {
        throw new sfValidatorError($this, 'max', array('value' => $value,
'max' => $this->getOption('max')));
    }
    if ($this->hasOption('min') && $clean < $this->getOption('min'))
    {
        throw new sfValidatorError($this, 'min', array('value' => $value,
'min' => $this->getOption('min')));
    }
    return $clean;
}
}
```

Check the symfony API documentation for widget and validator classes names and syntax.

Use options to pass parameters to the form class

A common issue with forms is to be able to use application parameters, such as the user's culture. The fastest but ugly way is to retrieve the user instance through the `sfContext` instance, using the `sfContext::getInstance()->getUser()` method. However, this solution creates a big coupling between the form and the context, making the testing and reusability more difficult. To avoid this problem, you can simply use option to pass the `culture` value to the form :

```
// from an action
public function executeContact(sfWebRequest $request)
{
    $this->form = new ContactForm(array(), array('culture' =>
$this->getUser()->getCulture()));
}

// from a unit test
$form = new ContactForm(array(), array('culture' => 'en'));

class ContactForm extends sfForm
{
    public function configure()
    {
        /* ... */
        $this->setWidget('country', new
sfWidgetFormI18NCountry(array('culture' => $this->getOption('culture'))));
        /* ... */
    }
}
```

Listing
10-47

Forms Based on a Model

Forms are the primary way to edit database records in web applications. And most forms in symfony applications allow the editing of a Model object. But the information necessary to build a form to edit a model already exists: it is in the schema. So symfony provides a form generator for model objects, that makes the creation of model-editing forms a snap.



Similar features to the ones described below exist for Doctrine.

Generating Model Forms

Symfony can deduce the widget types and the validators to use for a model editing form, based on the schema. Take the following schema, for instance with the Propel ORM:

```
// config/schema.yml
propel:
    article:
        id: ~
        title: { type: varchar(255), required: true }
        slug: { type: varchar(255), required: true, index: unique }
        content: longvarchar
        is_published: { type: boolean, required: true }
```

Listing
10-48

```

author_id: { type: integer, required: true, foreignTable: author,
foreignReference: id, OnDelete: cascade }
created_at: ~

author:
id: ~
first_name: varchar(20)
last_name: varchar(20)
email: { type: varchar(255), required: true, index: unique }
active: boolean

```

A form to edit an Article object should use a hidden widget for the id, a text widget for the title, a string validator for the title, etc. Symfony generates the form for you, provided that you call the `propel:build-forms` task:

Listing 10-49

```

// propel
$ php symfony propel:build-forms

// doctrine
$ php symfony doctrine:build-forms

```

For each table in the model, this command creates two files under the `lib/form/` directory: a `BaseXXXForm` class, overridden each time you call the `propel:build-forms` task, and an empty `XXXForm` class, extending the previous one. It is the same system as the Propel model classes generation.

The generated `lib/form/base/BaseArticleForm.class.php` contains the translation into widgets and validators of the columns defined for the `article` table in the `schema.yml`:

Listing 10-50

```

class BaseArticleForm extends BaseFormPropel
{
    public function setup()
    {
        $this->setWidgets(array(
            'id'          => new sfWidgetFormInputHidden(),
            'title'       => new sfWidgetFormInputText(),
            'slug'        => new sfWidgetFormInputText(),
            'content'     => new sfWidgetFormTextarea(),
            'is_published' => new sfWidgetFormInputCheckbox(),
            'author_id'   => new sfWidgetFormPropelChoice(array('model' =>
'Author', 'add_empty' => false)),
            'created_at'  => new sfWidgetFormDatetime(),
        ));
        $this->setValidators(array(
            'id'          => new sfValidatorPropelChoice(array('model' =>
'Article', 'column' => 'id', 'required' => false)),
            'title'       => new sfValidatorString(array('max_length' => 255)),
            'slug'        => new sfValidatorString(array('max_length' => 255)),
            'content'     => new sfValidatorString(array('max_length' => 255,
'required' => false)),
            'is_published' => new sfValidatorBoolean(),
            'author_id'   => new sfValidatorPropelChoice(array('model' =>
'Author', 'column' => 'id')),
            'created_at'  => new sfValidatorDatetime(array('required' =>
false)),
        ));
        $this->setPostValidator(
            new sfValidatorPropelUnique(array('model' => 'Article', 'column' =>

```

```

array('slug'))
);
$this->widgetSchema->setNameFormat('article[%s]');
parent::setup();
}

public function getModelName()
{
    return 'Article';
}
}

```

Notice that even though the `id` column is an Integer, symfony checks that the submitted `id` exists in the table using a `sfValidatorPropelChoice` validator. The form generator always sets the strongest validation rules, to ensure the cleanest data in the database.

Using Model Forms

You can customize generated form classes for your entire project by adding code to the empty `ArticleForm::configure()` method.

Here is an example of model form handling in an action. In this form, the `slug` validator is modified to make it optional, and the `author_id` widget is customized to display only a subset of authors - the 'active' ones.

```
// in lib/form/ArticleForm.class.php
public function configure()
{
    $this->getWidget('author_id')->setOption('criteria',
$this->getOption('criteria'));
    $this->getValidator('slug')->setOption('required', false);
}

// in modules/foo/actions/actions.class.php
public function executeEditArticle($request)
{
    $c = new Criteria();
    $c->add(AuthorPeer::ACTIVE, true);

    $this->form = new ArticleForm(
        ArticlePeer::retrieveByPk($request->getParameter('id')),
        array('criteria' => $c)
    );

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

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

*Listing
10-51*

Instead of setting default values through an associative array, Model forms use a Model object to initialize the widget values. To display an empty form, just pass a new Model object.

The form submission handling is greatly simplified by the fact that the form object has an embedded Model object. Calling `$this->form->save()` on a valid form updates the embedded Article object with the cleaned values and triggers the `save()` method on the Article object, as well as on the related objects if they exist.



The action code required to deal with a form is pretty much always the same, but that's not a reason to copy it from one module to the other. Symfony provides a module generator that creates the whole action and template code to manipulate a Model object through symfony forms.

Conclusion

The symfony form component is an entire framework on its own. It facilitates the display of forms in the view through widgets, it facilitates validation and handling of forms in the controller through validators, and it facilitates the edition of Model objects through Model forms. Although designed with a clear MVC separation, the form sub-framework is always easy to use. Most of the time, code generation will reduce your custom form code to a few lines.

There is much more in symfony form classes than what this chapter exposes. In fact, there is an entire book⁴² describing all its features through usage examples. And if the form framework itself doesn't provide the widget or the validator you need, it is designed in such an extensible way that you will only need to write a single class to get exactly what you need.

42. http://www.symfony-project.org/book/forms/1_4/en/

Chapter 11

Emails

Sending emails with symfony is simple and powerful, thanks to the usage of the Swift Mailer⁴³ library. Although Swift Mailer makes sending emails easy, symfony provides a thin wrapper on top of it to make sending emails even more flexible and powerful. This chapter will teach you how to put all the power at your disposal.



symfony 1.3 embeds Swift Mailer version 4.1.

Introduction

Email management in symfony is centered around a mailer object. And like many other core symfony objects, the mailer is a factory. It is configured in the `factories.yml` configuration file, and always available via the context instance:

```
$mailer = sfContext::getInstance()->getMailer();
```

*Listing
11-1*



Unlike other factories, the mailer is loaded and initialized on demand. If you don't use it, there is no performance impact whatsoever.

This tutorial explains the Swift Mailer integration in symfony. If you want to learn the nitty-gritty details of the Swift Mailer library itself, refer to its dedicated documentation⁴⁴.

Sending Emails from an Action

From an action, retrieving the mailer instance is made simple with the `getMailer()` shortcut method:

```
$mailer = $this->getMailer();
```

*Listing
11-2*

The Fastest Way

Sending an email is then as simple as using the `sfAction::composeAndSend()` method:

43. <http://www.swiftmailer.org/>

44. <http://www.swiftmailer.org/docs>

Listing 11-3

```
$this->getMailer()->composeAndSend(
    'from@example.com',
    'fabien@example.com',
    'Subject',
    'Body'
);
```

The `composeAndSend()` method takes four arguments:

- the sender email address (`from`);
- the recipient email address(es) (`to`);
- the subject of the message;
- the body of the message.

Whenever a method takes an email address as a parameter, you can pass a string or an array:

Listing 11-4

```
$address = 'fabien@example.com';
$address = array('fabien@example.com' => 'Fabien Potencier');
```

Of course, you can send an email to several people at once by passing an array of emails as the second argument of the method:

Listing 11-5

```
$to = array(
    'foo@example.com',
    'bar@example.com',
);
$this->getMailer()->composeAndSend('from@example.com', $to, 'Subject',
'Body');

$to = array(
    'foo@example.com' => 'Mr Foo',
    'bar@example.com' => 'Miss Bar',
);
$this->getMailer()->composeAndSend('from@example.com', $to, 'Subject',
'Body');
```

The Flexible Way

If you need more flexibility, you can also use the `sfAction::compose()` method to create a message, customize it the way you want, and eventually send it. This is useful, for instance, when you need to add an attachment as shown below:

Listing 11-6

```
// create a message object
$message = $this->getMailer()
    ->compose('from@example.com', 'fabien@example.com', 'Subject', 'Body')
    ->attach(Swift_Attachment::fromPath('/path/to/a/file.zip'))
;

// send the message
$this->getMailer()->send($message);
```

The Powerful Way

You can also create a message object directly for even more flexibility:

Listing 11-7

```
$message = Swift_Message::newInstance()
    ->setFrom('from@example.com')
```

```

->setTo('to@example.com')
->setSubject('Subject')
->setBody('Body')
->attach(Swift_Attachment::fromPath('/path/to/a/file.zip'))
;

$this->getMailer()->send($message);

```



The “Creating Messages”⁴⁵ and “Message Headers”⁴⁶ sections of the Swift Mailer official documentation describe all you need to know about creating messages.

Using the Symfony View

Sending your emails from your actions allows you to leverage the power of partials and components quite easily.

```
$message->setBody($this->getPartial('partial_name', $arguments));
```

*Listing
11-8*

Configuration

As any other symfony factory, the mailer can be configured in the `factories.yml` configuration file. The default configuration reads as follows:

```

mailer:
  class: sfMailer
  param:
    logging:          %SF_LOGGING_ENABLED%
    charset:         %SF_CHARSET%
    delivery_strategy: realtime
    transport:
      class: Swift_SmtpTransport
      param:
        host:      localhost
        port:      25
        encryption: ~
        username:  ~
        password:  ~

```

*Listing
11-9*

When creating a new application, the local `factories.yml` configuration file overrides the default configuration with some sensible defaults for the `prod`, `env`, and `test` environments:

```

test:
  mailer:
    param:
      delivery_strategy: none

dev:
  mailer:
    param:
      delivery_strategy: none

```

*Listing
11-10*

45. <http://swiftmailer.org/docs/messages>
 46. <http://swiftmailer.org/docs/headers>

The Delivery Strategy

One of the most useful feature of the Swift Mailer integration in symfony is the delivery strategy. The delivery strategy allows you to tell symfony how to deliver email messages and is configured via the `delivery_strategy` setting of `factories.yml`. The strategy changes the way the `send()` method behaves. Four strategies are available by default, which should suit all the common needs:

- `realtime`: Messages are sent in realtime.
- `single_address`: Messages are sent to a single address.
- `spool`: Messages are stored in a queue.
- `none`: Messages are simply ignored.

The `realtime` Strategy

The `realtime` strategy is the default delivery strategy, and the easiest to setup as there is nothing special to do.

Email messages are sent via the transport configured in the `transport` section of the `factories.yml` configuration file (see the next section for more information about how to configure the mail transport).

The `single_address` Strategy

With the `single_address` strategy, all messages are sent to a single address, configured via the `delivery_address` setting.

This strategy is really useful in the development environment to avoid sending messages to real users, but still allow the developer to check the rendered message in an email reader.



If you need to verify the original `to`, `cc`, and `bcc` recipients, they are available as values of the following headers: `X-Swift-To`, `X-Swift-Cc`, and `X-Swift-Bcc` respectively.

Email messages are sent via the same email transport as the one used for the `realtime` strategy.

The `spool` Strategy

With the `spool` strategy, messages are stored in a queue.

This is the best strategy for the production environment, as web requests do not wait for the emails to be sent.

The `spool` class is configured with the `spool_class` setting. By default, symfony comes bundled with three of them:

- `Swift_FileSpool`: Messages are stored on the filesystem.
- `Swift DoctrineSpool`: Messages are stored in a Doctrine model.
- `Swift PropelSpool`: Messages are stored in a Propel model.

When the spool is instantiated, the `spool_arguments` setting is used as the constructor arguments. Here are the options available for the built-in queues classes:

- `Swift_FileSpool`:
 - The absolute path of the queue directory (messages are stored in this directory)

- **Swift_DoctrineSpool:**
 - The Doctrine model to use to store the messages (`MailMessage` by default)
 - The column name to use for message storage (`message` by default)
 - The method to call to retrieve the messages to send (optional). It receives the queue options as a argument.
- **Swift_PropelSpool:**
 - The Propel model to use to store the messages (`MailMessage` by default)
 - The column name to use for message storage (`message` by default)
 - The method to call to retrieve the messages to send (optional). It receives the queue options as a argument.

Here is a classic configuration for a Doctrine spool:

```
# Schema configuration in schema.yml
MailMessage:
  actAs: { Timestampable: ~ }
  columns:
    message: { type: clob, notnull: true }
```

*Listing
11-11*

```
# configuration in factories.yml
mailer:
  class: sfMailer
  param:
    delivery_strategy: spool
    spool_class:      Swift_DoctrineSpool
    spool_arguments:  [ MailMessage, message, getSpooledMessages ]
```

*Listing
11-12*

And the same configuration for a Propel spool:

```
# Schema configuration in schema.yml
mail_message:
  message: { type: clob, required: true }
  created_at: ~
```

*Listing
11-13*

```
# configuration in factories.yml
dev:
  mailer:
    param:
      delivery_strategy: spool
      spool_class:      Swift_PropelSpool
      spool_arguments:  [ MailMessage, message, getSpooledMessages ]
```

*Listing
11-14*

To send the message stored in a queue, you can use the `project:send-emails` task (note that this task is totally independent of the queue implementation, and the options it takes):

```
$ php symfony project:send-emails
```

*Listing
11-15*



The `project:send-emails` task takes an `application` and `env` options.

When calling the `project:send-emails` task, email messages are sent via the same transport as the one used for the `realtime` strategy.



Note that the `project:send-emails` task can be run on any machine, not necessarily on the machine that created the message. It works because everything is stored in the message object, even the file attachments.



The built-in implementation of the queues are very simple. They send emails without any error management, like they would have been sent if you have used the `realtime` strategy. Of course, the default queue classes can be extended to implement your own logic and error management.

The `project:send-emails` task takes two optional options:

- `message-limit`: Limits the number of messages to sent.
- `time-limit`: Limits the time spent to send messages (in seconds).

Both options can be combined:

```
$ php symfony project:send-emails --message-limit=10 --time-limit=20
```

The above command will stop sending messages when 10 messages are sent or after 20 seconds.

Even when using the `spool` strategy, you might need to send a message immediately without storing it in the queue. This is possible by using the special `sendNextImmediately()` method of the mailer:

Listing 11-16 `$this->getMailer()->sendNextImmediately()->send($message);`

In the previous example, the `$message` won't be stored in the queue and will be sent immediately. As its name implies, the `sendNextImmediately()` method only affects the very next message to be sent.



The `sendNextImmediately()` method has no special effect when the delivery strategy is not `spool`.

The none Strategy

This strategy is useful in the development environment to avoid emails to be sent to real users. Messages are still available in the web debug toolbar (more information in the section below about the mailer panel of the web debug toolbar).

It is also the best strategy for the test environment, where the `sfTesterMailer` object allows you to introspect the messages without the need to actually send them (more information in the section below about testing).

The Mail Transport

Mail messages are actually sent by a transport. The transport is configured in the `factories.yml` configuration file, and the default configuration uses the SMTP server of the local machine:

Listing 11-17 `transport:`
 `class: Swift_SmtpTransport`
 `param:`
 `host: localhost`
 `port: 25`

```
encryption: ~
username: ~
password: ~
```

Swift Mailer comes bundled with three different transport classes:

- `Swift_SmtpTransport`: Uses a SMTP server to send messages.
- `Swift_SendmailTransport`: Uses `sendmail` to send messages.
- `Swift_MailTransport`: Uses the native PHP `mail()` function to send messages.



The “Transport Types”⁴⁷ section of the Swift Mailer official documentation describes all you need to know about the built-in transport classes and their different parameters.

Sending an Email from a Task

Sending an email from a task is quite similar to sending an email from an action, as the task system also provides a `getMailer()` method.

When creating the mailer, the task system relies on the current configuration. So, if you want to use a configuration from a specific application, you must accept the `--application` option (see the chapter on tasks for more information on this topic).

Notice that the task uses the same configuration as the controllers. So, if you want to force the delivery when the `spool` strategy is used, use `sendNextImmediately()`:

```
$this->getMailer()->sendNextImmediately()->send($message);
```

*Listing
11-18*

Debugging

Traditionally, debugging emails has been a nightmare. With symfony, it is very easy, thanks to the web debug toolbar.

From the comfort of your browser, you can easily and rapidly see how many messages have been sent by the current action:



If you click on the email icon, the sent messages are displayed in the panel in their raw form as shown below.

47. <http://swiftmailer.org/docs/transport-types>



```

Sf 1.3.0-DEV config view logs 5632.0 KB 1532 ms 1 2 ×
Emails

Configuration
Delivery strategy: spool

Email sent
Subject (to: fabien.potencier@symfony-project.com) ↴

Message-ID: <1257158107.4aeeb5dbe396c@fabien.localhost>
Date: Mon, 02 Nov 2009 11:35:07 +0100
Subject: Subject
From: john@doe.com
To: fabien.potencier@symfony-project.com
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable
X-Priority: 1 (Highest)

Body

```



Each time an email is sent, symfony also adds a message in the log.

Testing

Of course, the integration would not have been complete without a way to test mail messages. By default, symfony registers a mailer tester (`sfMailerTester`) to ease mail testing in functional tests.

The `hasSent()` method tests the number of messages sent during the current request:

Listing 11-19

```
$browser->
    get('/foo')->
    with('mailer')->
        hasSent(1)
;
```

The previous code checks that the `/foo` URL sends only one email.

Each sent email can be further tested with the help of the `checkHeader()` and `checkBody()` methods:

Listing 11-20

```
$browser->
    get('/foo')->
    with('mailer')->begin()->
        hasSent(1)->
        checkHeader('Subject', '/Subject/')->
        checkBody('/Body/')->
    end()
;
```

The second argument of `checkHeader()` and the first argument of `checkBody()` can be one of the following:

- a string to check an exact match;
- a regular expression to check the value against it;
- a negative regular expression (a regular expression starting with a `!`) to check that the value does not match.

By default, the checks are done on the first message sent. If several messages have been sent, you can choose the one you want to test with the `withMessage()` method:

```
$browser->
    get('/foo')->
    with('mailer')->begin()->
        hasSent(2)->
        withMessage('foo@example.com')->
        checkHeader('Subject', '/Subject/')->
        checkBody('/Body/')->
    end()
;
```

Listing 11-21

The `withMessage()` takes a recipient as its first argument. It also takes a second argument to indicate which message you want to test if several ones have been sent to the same recipient.

Last but not the least, the `debug()` method dumps the sent messages to spot problems when a test fails:

```
$browser->
    get('/foo')->
    with('mailer')->
    debug()
;
```

Listing 11-22

Email Messages as Classes

In this chapter's introduction, you have learnt how to send emails from an action. This is probably the easiest way to send emails in a symfony application and probably the best when you just need to send a few simple messages.

But when your application needs to manage a large number of different email messages, you should probably have a different strategy.



As an added bonus, using classes for email messages means that the same email message can be used in different applications; a frontend and a backend one for instance.

As messages are plain PHP objects, the obvious way to organize your messages is to create one class for each of them:

```
// lib/email/ProjectConfirmationMessage.class.php
class ProjectConfirmationMessage extends Swift_Message
{
    public function __construct()
    {
        parent::__construct('Subject', 'Body');

        $this
            ->setFrom(array('app@example.com' => 'My App Bot'))
            ->attach('...');
    }
}
```

Listing 11-23

Sending a message from an action, or from anywhere else for that matter, is simple a matter of instantiating the right message class:

Listing 11-24 `$this->getMailer()->send(new ProjectConfirmationMessage());`

Of course, adding a base class to centralize the shared headers like the `From` header, or to add a common signature can be convenient:

Listing 11-25 `// lib/email/ProjectConfirmationMessage.class.php
class ProjectConfirmationMessage extends ProjectBaseMessage
{
 public function __construct()
 {
 parent::__construct('Subject', 'Body');

 // specific headers, attachments, ...
 $this->attach('...');
 }
}

// lib/email/ProjectBaseMessage.class.php
class ProjectBaseMessage extends Swift_Message
{
 public function __construct($subject, $body)
 {
 $body .= <<<EOF
--
Email sent by My App Bot
EOF
;
 parent::__construct($subject, $body);

 // set all shared headers
 $this->setFrom(array('app@example.com' => 'My App Bot'));
 }
}`

If a message depends on some model objects, you can of course pass them as arguments to the constructor:

Listing 11-26 `// lib/email/ProjectConfirmationMessage.class.php
class ProjectConfirmationMessage extends ProjectBaseMessage
{
 public function __construct($user)
 {
 parent::__construct('Confirmation for '.$user->getName(), 'Body');
 }
}`

Recipes

Sending Emails via Gmail

If you don't have an SMTP server but have a Gmail account, use the following configuration to use the Google servers to send and archive messages:

```
transport:
  class: Swift_SmtpTransport
  param:
    host:      smtp.gmail.com
    port:      465
    encryption: ssl
    username:  your_gmail_username_goes_here
    password:  your_gmail_password_goes_here
```

*Listing
11-27*

Replace the `username` and `password` with your Gmail credentials and you are done.

Customizing the Mailer Object

If configuring the mailer via the `factories.yml` is not enough, you can listen to the `mailer.configure` event, and further customize the mailer.

You can connect to this event in your `ProjectConfiguration` class like shown below:

```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    // ...

    $this->dispatcher->connect(
      'mailer.configure',
      array($this, 'configureMailer')
    );
  }

  public function configureMailer(sfEvent $event)
  {
    $mailer = $event->getSubject();

    // do something with the mailer
  }
}
```

*Listing
11-28*

The following section illustrates a powerful usage of this technique.

Using Swift Mailer Plugins

To use Swift Mailer plugins, listen to the `mailer.configure` event (see the section above):

```
public function configureMailer(sfEvent $event)
{
  $mailer = $event->getSubject();

  $plugin = new Swift_Plugins_ThrottlerPlugin(
```

*Listing
11-29*

```

    100, Swift_Plugins_ThrottlerPlugin::MESSAGES_PER_MINUTE
);

$mailer->registerPlugin($plugin);
}

```



The “Plugins”⁴⁸ section of the Swift Mailer official documentation describes all you need to know about the built-in plugins.

Customizing the Spool Behavior

The built-in implementation of the spools is very simple. Each spool retrieves all emails from the queue in a random order and sends them.

You can configure a spool to limit the time spent to send emails (in seconds), or to limit the number of messages to send:

Listing 11-30

```

$spool = $mailer->getSpool();

$spool->setMessageLimit(10);
$spool->setTimeLimit(10);

```

In this section, you will learn how to implement a priority system for the queue. It will give you all the information needed to implement your own logic.

First, add a `priority` column to the schema:

Listing 11-31

```

# for Propel
mail_message:
    message: { type: clob, required: true }
    created_at: ~
    priority: { type: integer, default: 3 }

# for Doctrine
MailMessage:
    actAs: { Timestampable: ~ }
    columns:
        message: { type: clob, notnull: true }
        priority: { type: integer }

```

When sending an email, set the priority header (1 means highest):

Listing 11-32

```

$message = $this->getMailer()
    ->compose('john@doe.com', 'foo@example.com', 'Subject', 'Body')
    ->setPriority(1)
;
$this->getMailer()->send($message);

```

Then, override the default `setMessage()` method to change the priority of the `MailMessage` object itself:

Listing 11-33

```

// for Propel
class MailMessage extends BaseMailMessage
{
    public function setMessage($message)

```

48. <http://swiftmailer.org/docs/plugins>

```

    {
        $msg = unserialize($message);
        $this->setPriority($msg->getPriority());

        parent::setMessage($message);
    }
}

// for Doctrine
class MailMessage extends BaseMailMessage
{
    public function setMessage($message)
    {
        $msg = unserialize($message);
        $this->priority = $msg->getPriority();

        $this->_set('message', $message);
    }
}

```

Notice that the message is serialized by the queue, so it has to be unserialized before getting the priority value. Now, create a method that orders the messages by priority:

```

// for Propel
class MailMessagePeer extends BaseMailMessagePeer
{
    static public function getSpooledMessages(Criteria $criteria)
    {
        $criteria->addAscendingOrderByColumn(self::PRIORITY);

        return self::doSelect($criteria);
    }

    // ...
}

// for Doctrine
class MailMessageTable extends Doctrine_Table
{
    public function getSpooledMessages()
    {
        return $this->createQuery('m')
            ->orderBy('m.priority')
            ;
    }

    // ...
}

```

*Listing
11-34*

The last step is to define the retrieval method in the `factories.yml` configuration to change the default way in which the messages are obtained from the queue:

```
spool_arguments: [ MailMessage, message, getSpooledMessages ]
```

*Listing
11-35*

That's all there is to it. Now, each time you run the `project:send-emails` task, each email will be sent according to its priority.

Customizing the Spool with any Criteria

The previous example uses a standard message header, the priority. But if you want to use any criteria, or if you don't want to alter the sent message, you can also store the criteria as a custom header, and remove it before sending the email.

First, add a custom header to the message to be sent:

```
Listing 11-36 public function executeIndex()
{
    $message = $this->getMailer()
        ->compose('john@doe.com', 'foo@example.com', 'Subject', 'Body')
        ;

    $message->getHeaders()->addTextHeader('X-Queue-Criteria', 'foo');

    $this->getMailer()->send($message);
}
```

Then, retrieve the value from this header when storing the message in the queue, and remove it immediately:

```
Listing 11-37 public function setMessage($message)
{
    $msg = unserialize($message);

    $headers = $msg->getHeaders();
    $criteria = $headers->get('X-Queue-Criteria')->getFieldBody();
    $this->setCriteria($criteria);
    $headers->remove('X-Queue-Criteria');

    parent::setMessage($message);
}
```

Chapter 12

Caching

One of the ways to speed up an application is to store chunks of generated HTML code, or even full pages, for future requests. This technique is known as caching, and it can be managed on the server side and on the client side.

Symfony offers a flexible server-caching system. It allows saving the full response, the result of an action, a partial, or a template fragment into a file, through a very intuitive setup based on YAML files. When the underlying data changes, you can easily clear selective parts of the cache with the command line or special action methods. Symfony also provides an easy way to control the client-side cache through HTTP 1.1 headers. This chapter deals with all these subjects, and gives you a few tips on monitoring the improvements that caching can bring to your applications.

Caching the Response

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. 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 no cached version is 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 enable it in order to improve performance.

Symfony handles three different types of HTML caching:

- Cache of an action (with or without the layout)
- Cache of a partial or a component
- Cache of a template fragment

The first two types are handled with YAML configuration files. Template fragment caching is managed by calls to helper functions in the template.

Global Cache Settings

For each application of a project, the HTML cache mechanism can be enabled or disabled (the default), per environment, in the `cache` setting of the `settings.yml` file. Listing 12-1 demonstrates enabling the ability to cache. Note this will not cause anything to be cached; to do so you must enable caching in each module/action (see below).

Listing 12-1 - Activating the Cache, in frontend/config/settings.yml

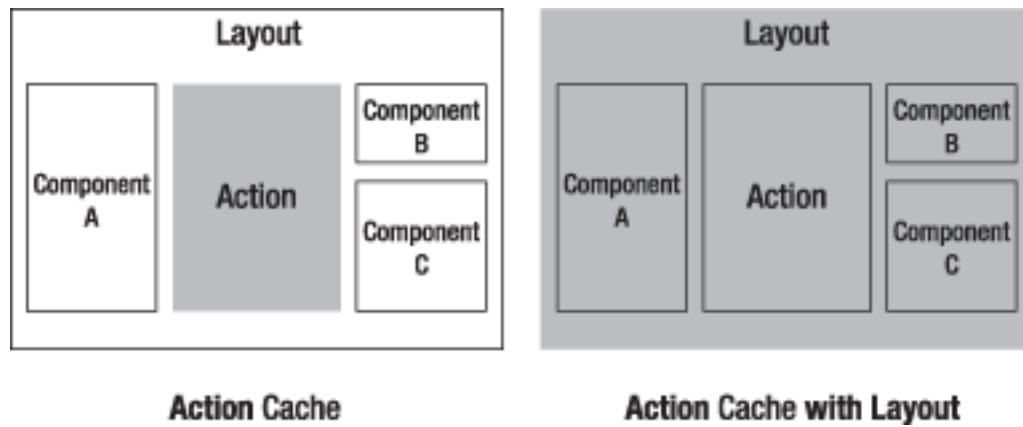
Listing 12-1

```
dev:
  .settings:
    cache: true
```

Caching an Action

Actions displaying static information (not depending on database or session-dependent data) or actions reading information from a database but without modifying it (typically, GET requests) are often ideal for caching. Figure 12-1 shows which elements of the page are cached in this case: either the action result (its template) or the action result together with the layout.

Figure 12-1 - Caching an action



Action Cache

Action Cache with Layout

For instance, consider a `user/list` action that returns the list of all users of a website. Unless a user is modified, added, or removed (and this matter will be discussed later in the “Removing Items from the Cache” section), this list always displays the same information, so it is a good candidate for caching.

Cache activation and settings, action by action, are defined in a `cache.yml` file located in the module `config/` directory. See Listing 12-2 for an example.

Listing 12-2 - Activating the Cache for an Action, in `frontend/modules/user/config/cache.yml`

Listing 12-2

```
list:
  enabled:      true
  with_layout: false  # Default value
  lifetime:    86400  # Default value
```

This configuration stipulates that the cache is on for the `list` action, and that the layout will not be cached with the action (which is the default behavior). It means that even if a cached version of the action exists, the layout (together with its partials and components) is still executed. If the `with_layout` setting is set to `true`, the layout is cached with the action and not executed again.

To test the cache settings, call the action in the development environment from your browser.

Listing 12-3

```
http://myapp.example.com/frontend\_dev.php/user/list
```

You will notice a border around the action area in the page. The first time, the area has a blue header, showing that it did not come from the cache. Refresh the page, and the action area will have a yellow header, showing that it did come from the cache (with a notable boost in response time). You will learn more about the ways to test and monitor caching later in this chapter.



Slots are part of the template, and caching an action will also store the value of the slots defined in this action's template. So the cache works natively for slots.

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. Modify the `cache.yml` file to enable the cache for this action as well, as shown in Listing 12-3.

In order to organize your `cache.yml`, you can regroup the settings for all the actions of a module under the `all:` key, also shown in Listing 12-3.

Listing 12-3 - A Full cache.yml Example, in frontend/modules/user/config/cache.yml

```
list:  
  enabled: true  
show:  
  enabled: true  
  
all:  
  with_layout: false # Default value  
  lifetime: 86400 # Default value
```

Listing 12-4

Now, every call to the `user/show` action with a different `id` argument creates a new record in the cache. So the cache for this:

`http://myapp.example.com/user/show/id/12`

Listing 12-5

will be different than the cache for this:

`http://myapp.example.com/user/show/id/25`

Listing 12-6



Actions called with a POST method or with GET parameters are not cached.

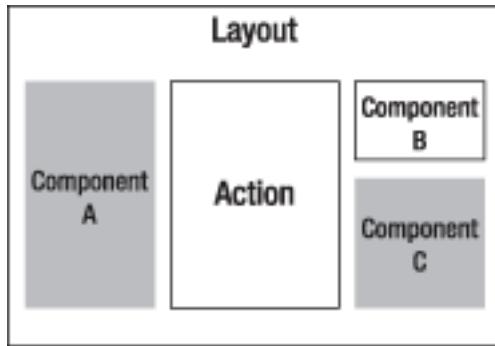
The `with_layout` setting deserves a few more words. It actually determines what kind of data is stored in the cache. For the cache without layout, only the result of the template execution and the action variables are stored in the cache. For the cache with layout, the whole response object is stored. This means that the cache with layout is much faster than the cache without it.

If you can functionally afford it (that is, if the layout doesn't rely on session-dependent data), you should opt for the cache with layout. Unfortunately, the layout often contains some dynamic elements (for instance, the name of the user who is connected), so action cache without layout is the most common configuration. However, RSS feeds, pop-ups, and pages that don't depend on cookies can be cached with their layout.

Caching a Partial or Component

Chapter 7 explained how to reuse code fragments across several templates, using the `include_partial()` helper. A partial is as easy to cache as an action, and its cache activation follows the same rules, as shown in Figure 12-2.

Figure 12-2 - Caching a partial or component



Component Cache

For instance, Listing 12-4 shows how to edit the `cache.yml` file to enable the cache on a `_my_partial.php` partial located in the `user` module. Note that the `with_layout` setting doesn't make sense in this case.

Listing 12-4 - Caching a Partial, in `frontend/modules/user/config/cache.yml`

```

Listing 12-7
_my_partial:
  enabled: true
list:
  enabled: true
...
  
```

Now all the templates using this partial won't actually execute the PHP code of the partial, but will use the cached version instead.

```

Listing 12-8
<?php include_partial('user/my_partial') ?>
  
```

Just as for actions, partial caching is also relevant when the result of the partial depends on parameters. The cache system will store as many versions of a template as there are different values of parameters.

```

Listing 12-9
<?php include_partial('user/my_other_partial', array('foo' => 'bar')) ?>
  
```



The action cache is more powerful than the partial cache, since when an action is cached, the template is not even executed; if the template contains calls to partials, these calls are not performed. Therefore, partial caching is useful only if you don't use action caching in the calling action or for partials included in the layout.

A little reminder from Chapter 7: A component is a light action put on top of a partial. This inclusion type is very similar to partials, and support caching in the same way. For instance, if your global layout includes a component called `day` with `include_component('general/day')` in order to show the current date, set the `cache.yml` file of the `general` module as follows to enable the cache on this component:

```

Listing 12-10
_day:
  enabled: true
  
```

When caching a component or a partial, you must decide whether to store a single version for all calling templates or a version for each template. By default, a component is stored independently of the template that calls it. But contextual components, such as a component that displays a different sidebar with each action, should be stored as many times as there are templates calling it. The caching system can handle this case, provided that you set the `contextual` parameter to `true`, as follows:

```
_day:
contextual: true
enabled: true
```

*Listing
12-11*

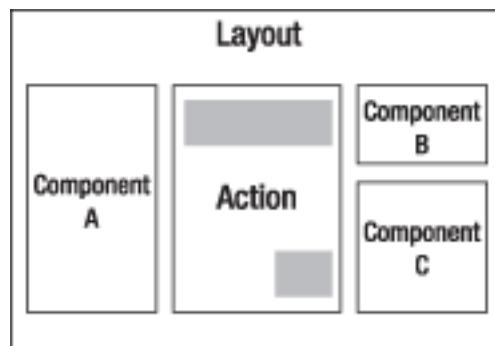


Global components (the ones located in the application `templates/` directory) can be cached, provided that you declare their cache settings in the application `cache.yml`.

Caching a Template Fragment

Action caching applies to only a subset of actions. For the other actions—those that update data or display session-dependent information in the template—there is still room for cache improvement but in a different way. Symfony provides a third cache type, which is dedicated to template fragments and enabled directly inside the template. In this mode, the action is always executed, and the template is split into executed fragments and fragments in the cache, as illustrated in Figure 12-3.

Figure 12-3 - Caching a template fragment



Fragment Cache

For instance, you may have a list of users that shows a link of the last-accessed user, and this information is dynamic. The `cache()` helper defines the parts of a template that are to be put in the cache. See Listing 12-5 for details on the syntax.

Listing 12-5 - Using the `cache()` Helper, in `frontend/modules/user/templates/listSuccess.php`

```
<!-- Code executed each time -->
<?php echo link_to('last accessed user', 'user/
show?id='.$last_accessed_user_id) ?>

<!-- Cached code -->
<?php if (!cache('users')): ?>
    <?php foreach ($users as $user): ?>
        <?php echo $user->getName() ?>
    <?php endforeach; ?>
    <?php cache_save() ?>
<?php endif; ?>
```

*Listing
12-12*

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, the code between these lines is processed and saved in the cache, identified with the unique fragment name.



The code not included between such lines is always processed and not cached.



The action (`list` in the example) must not have caching enabled, since this would bypass the whole template execution and ignore the fragment cache declaration.

The speed boost of using the template fragment cache is not as significant as with the action cache, since the action is always executed, the template is partially processed, and the layout is always used for decoration.

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.

As with actions and components, cached fragments can take a lifetime in seconds as a second argument of the call to the `cache()` helper.

Listing 12-13 <?php if (!cache('users', 43200)): ?>

The default cache lifetime (86400 seconds, or one day) is used if no parameter is given to the helper.



Another way to make an action cacheable is to insert the variables that make it vary into the action's routing pattern. For instance, if a home page displays the name of the connected user, it cannot be cached unless the URL contains the user nickname. Another example is for internationalized applications: If you want to enable caching on a page that has several translations, the language code must somehow be included in the URL pattern. This trick will multiply the number of pages in the cache, but it can be of great help to speed up heavily interactive applications.

Configuring the Cache Dynamically

The `cache.yml` file is one way to define cache settings, but it has the inconvenience of being invariant. However, as usual in symfony, you can use plain PHP rather than YAML, and that allows you to configure the cache dynamically.

Why would you want to change the cache settings dynamically? A good example is a page that is different for authenticated users and for anonymous ones, but the URL remains the same. Imagine an `article/show` page with a rating system for articles. The rating feature is disabled for anonymous users. For those users, rating links trigger the display of a login form. This version of the page can be cached. On the other hand, for authenticated users, clicking a rating link makes a POST request and creates a new rating. This time, the cache must be disabled for the page so that symfony builds it dynamically.

The right place to define dynamic cache settings is in a filter executed before the `sfCacheFilter`. Indeed, the cache is a filter in symfony, just like the security features. In order to enable the cache for the `article/show` page only if the user is not authenticated, create a `conditionalCacheFilter` in the application `lib/` directory, as shown in Listing 12-6.

Listing 12-6 - Configuring the Cache in PHP, in frontend/lib/conditionalCacheFilter.class.php

Listing 12-14

```
class conditionalCacheFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $context = $this->getContext();
        if (!$context->getUser()->isAuthenticated())
        {
```

```

        foreach ($this->getParameter('pages') as $page)
        {
            $context->getViewCacheManager()->addCache($page['module'],
$page['action'], array('lifeTime' => 86400));
        }
    }

    // Execute next filter
    $filterChain->execute();
}
}

```

You must register this filter in the `filters.yml` file before the `sfCacheFilter`, as shown in Listing 12-7.

Listing 12-7 - Registering Your Custom Filter, in frontend/config/filters.yml

```

...
security: ~

conditionalCache:
    class: conditionalCacheFilter
    param:
        pages:
            - { module: article, action: show }

cache: ~
...

```

*Listing
12-15*

Clear the cache (to autoload the new filter class), and the conditional cache is ready. It will enable the cache of the pages defined in the `pages` parameter only for users who are not authenticated.

The `addCache()` method of the `sfViewCacheManager` object expects a module name, an action name, and an associative array with the same parameters as the ones you would define in a `cache.yml` file. For instance, if you want to define that the `article/show` action must be cached with the layout and with a lifetime of 3600 seconds, then write the following:

```
$context->getViewCacheManager()->addCache('article', 'show', array(
    'withLayout' => true,
    'lifeTime'   => 3600,
));
```

*Listing
12-16*

Alternative Caching storage

By default, the symfony cache system stores data in files on the web server hard disk. You may want to store cache in memory (for instance, via memcached⁴⁹) or in a database (notably if you want to share your cache among several servers or speed up cache removal). You can easily alter symfony's default cache storage system because the cache class used by the symfony view cache manager is defined in `factories.yml`.

The default view cache storage factory is the `sfFileCache` class:

```
Listing 12-17    view_cache:  
               class: sfFileCache  
               param:  
                 automaticCleaningFactor: 0  
                 cacheDir: %SF_TEMPLATE_CACHE_DIR%
```

You can replace the `class` with your own cache storage class or with one of the symfony alternative classes (including `sfAPCCache`, `sfEAcceleratorCache`, `sfMemcacheCache`, `sfSQLiteCache`, and `sfXCacheCache`). The parameters defined under the `param` key are passed to the constructor of the cache class as an associative array. Any view cache storage class must implement all methods found in the abstract `sfCache` class. Refer to the Chapter 19 for more information on this subject.

Configuration for a memcache backend using two memcache servers: `view_cache: class: sfMemcacheCache param: servers: server_1: host: 192.168.1.10 server_2: host: 192.168.1.11`

Using the Super Fast Cache

Even a cached page involves some PHP code execution. For such a page, symfony still loads the configuration, builds the response, and so on. 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 without a suffix or with `.html`.

You can do this by hand, page by page, with a simple command-line call:

```
Listing 12-18 $ curl http://myapp.example.com/user/list.html > web/user/list.html
```

After that, every time that the `user/list` action is requested, Apache finds the corresponding `list.html` page and bypasses symfony completely. The trade-off is that you can't control the page cache with symfony anymore (lifetime, automatic deletion, and so on), but the speed gain is very impressive.

Alternatively, you can use the `sfSuperCache` symfony plug-in, which automates the process and supports lifetime and cache clearing. Refer to Chapter 17 for more information about plug-ins.

49. <http://www.danga.com/memcached/>

Other Speedup tactics

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.

PHP accelerators (eAccelerator, APC, XCache, and so on), also called opcode caching modules, increase performance of PHP scripts by caching them in a compiled state, so that the overhead of code parsing and compiling is almost completely eliminated. This is particularly effective for the ORMs classes, which contain a great amount of code. These accelerators are compatible with symfony and can easily triple the speed of an application. They are recommended in production environments for any symfony application with a large audience.

With a PHP accelerator, you can manually store persistent data in memory, to avoid doing the same processing for each request, with the `sfAPCCache` class if use APC for instance. And if you want to cache the result of a CPU-intensive function, you will probably use the `sfFunctionCache` object. Refer to Chapter 18 for more information about these mechanisms.

Removing Items from the Cache

If the scripts or the data of your application change, the cache will contain outdated information. To avoid incoherence and bugs, you can remove parts of the cache in many different ways, according to your needs.

Clearing the Entire Cache

The `cache:clear` task of the symfony command line erases the cache (HTML, configuration, routing, and i18n cache). You can pass it arguments to erase only a subset of the cache, as shown in Listing 12-8. Remember to call it only from the root of a symfony project.

Listing 12-8 - Clearing the Cache

```
// Erase the whole cache
$ php symfony cache:clear

// Short syntax
$ php symfony cc

// Erase only the cache of the frontend application
$ php symfony cache:clear --app=frontend

// Erase only the HTML cache of the frontend application
$ php symfony cache:clear --app=frontend --type=template

// Erase only the configuration cache of the frontend application
// The built-types are config, i18n, routing, and template.
$ php symfony cache:clear --app=frontend --type=config

// Erase only the configuration cache of the frontend application and the
// prod environment
$ php symfony cache:clear --app=frontend --type=config --env=prod
```

*Listing
12-19*

Clearing Selective Parts of the Cache

When the database is updated, the cache of the actions related to the modified data must be cleared. You could clear the whole cache, but that would be a waste for all the existing cached actions that are unrelated to the model change. This is where the `remove()` method of the `sfViewCacheManager` object applies. It expects an internal URI as argument (the same kind of argument you would provide to a `link_to()`), and removes the related action cache.

For instance, imagine that the `update` action of the `user` module modifies the columns of a `User` object. The cached versions of the `list` and `show` actions need to be cleared, or else the old versions, which contain erroneous data, are displayed. To handle this, use the `remove()` method, as shown in Listing 12-9.

Listing 12-9 - Clearing the Cache for a Given Action, in `modules/user/actions/actions.class.php`

```
Listing 12-20
public function executeUpdate($request)
{
    // Update a user
    $user_id = $request->getParameter('id');
    $user = UserPeer::retrieveByPk($user_id);
    $this->forward404Unless($user);
    $user->setName($request->getParameter('name'));
    ...
    $user->save();

    // Clear the cache for actions related to this user
    $cacheManager = $this->getContext()->getViewCacheManager();
    $cacheManager->remove('user/list');
    $cacheManager->remove('user/show?id=' . $user_id);
    ...
}
```

Removing cached partials and components is a little trickier. As you can pass them any type of parameter (including objects), it is almost impossible to identify their cached version afterwards. Let's focus on partials, as the explanation is the same for the other template components. Symfony identifies a cached partial with a special prefix (`sf_cache_partial`), the name of the module, and the name of the partial, plus a hash of all the parameters used to call it, as follows:

```
Listing 12-21
// A partial called by
<?php include_partial('user/my_partial', array('user' => $user) ?>

// Is identified in the cache as
@sf_cache_partial?module=user&action=_my_partial
    &sf_cache_key=bf41dd9c84d59f3574a5da244626dcc8
```

In theory, you could remove a cached partial with the `remove()` method if you knew the value of the parameters hash used to identify it, but this is very impracticable. Fortunately, if you add a `sf_cache_key` parameter to the `include_partial()` helper call, you can identify the partial in the cache with something that you know. As you can see in Listing 12-10, clearing a single cached partial—for instance, to clean up the cache from the partial based on a modified `User`—becomes easy.

Listing 12-10 - Clearing Partials from the Cache

```
Listing 12-22
<?php include_partial('user/my_partial', array(
    'user'          => $user,
```

```
'sf_cache_key' => $user->getId()  
)?>  
  
// Is identified in the cache as  
@sf_cache_partial?module=user&action=_my_partial&sf_cache_key=12  
  
// Clear _my_partial for a specific user in the cache with  
$cacheManager->remove('@sf_cache_partial?module=user&action=_my_partial  
&sf_cache_key='.$user->getId());
```

To clear template fragments, use the same `remove()` method. The key identifying the fragment in the cache is composed of the same `sf_cache_partial` prefix, the module name, the action name, and the `sf_cache_key` (the unique name of the cache fragment included by the `cache()` helper). Listing 12-11 shows an example.

Listing 12-11 - Clearing Template Fragments from the Cache

```
<!-- Cached code -->  
<?php if (!cache('users')): ?>  
... // Whatever  
<?php cache_save() ?>  
<?php endif; ?>  
  
// Is identified in the cache as  
@sf_cache_partial?module=user&action=list&sf_cache_key=users  
  
// Clear it with  
$cacheManager->remove('@sf_cache_partial?module=user&action=list  
&sf_cache_key=users');
```

*Listing
12-23*

Selective Cache Clearing Can damage your Brain

The trickiest part of the cache-clearing job is to determine which actions are influenced by a data update.

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 descriptions of the user's publications and the list of publications. This means that you need to add to the `update` action of the `user` module, something like this:

```
Listing 12-24 $c = new Criteria();
$c->add(PublicationPeer::AUTHOR_ID, $request->getParameter('id'));
$publications = PublicationPeer::doSelect($c);

$cacheManager = sfContext::getInstance()->getViewCacheManager();
foreach ($publications as $publication)
{
    $cacheManager->remove('publication/show?id=' . $publication->getId());
}
$cacheManager->remove('publication/list');
```

When you start using the HTML cache, you need to keep a clear view of the dependencies between the model and 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.

And, if you don't want to damage your brain with too difficult an analysis, you can always clear the whole cache each time you update the database...

Clearing several cache parts at once

The `remove()` method accepts keys with wildcards. It allows you to remove several cache parts with a single call. You can do for instance:

```
Listing 12-25 $cacheManager->remove('user/show?id=*'); // Remove for all user records
```

Another good example is with applications handling several languages, where the language code appears in all URLs. The URL to a user profile page should look like this:

```
Listing 12-26 http://www.myapp.com/en/user/show/id/12
```

To remove the cached profile of the user having an `id` of 12 in all languages, you can simply call:

```
Listing 12-27 $cache->remove('user/show?sf_culture=*&id=12');
```

This also works for partials:

```
Listing 12-28 $cacheManager->remove('@sf_cache_partial?module=user&action=_my_partial
&sf_cache_key=*'); // Remove for all keys
```

The `remove()` method accepts two additional parameters, allowing you to define which hosts and vary headers you want to clear the cache for. This is because symfony keeps one cache version for each host and vary headers, so that two applications sharing the same code base but not the same hostname use different caches. This can be of great use, for instance, when

an application interprets the subdomain as a request parameter (like `http://php.asksheet.com` and `http://life.asksheet.com`). If you don't set the last two parameters, symfony will remove the cache for the current host and for the `all` vary header. Alternatively, if you want to remove the cache for another host, call `remove()` as follows:

```
$cacheManager->remove('user/show?id=*'); // Remove
records for the current host and all users
$cacheManager->remove('user/show?id=*', 'life.asksheet.com'); // Remove
records for the host life.asksheet.com and all users
$cacheManager->remove('user/show?id=*', '*'); // Remove
records for every host and all users
```

*Listing
12-29*

The `remove()` method works in all the caching strategies that you can define in the `factories.yml` (not only `sfFileCache`, but also `sfAPCCache`, `sfEAcceleratorCache`, `sfMemcacheCache`, `sfSQLiteCache`, and `sfXCacheCache`).

Clearing cache across applications

Clearing the cache across applications can be a problem. For instance, if an administrator modifies a record in the `user` table in a `backend` application, all the actions depending on this user in the `frontend` application need to be cleared from the cache. But the view cache manager available in the `backend` application doesn't know the `frontend` application routing rules (applications are isolated from each other). So you can't write this code in the `backend`:

```
$cacheManager = sfContext::getInstance()->getViewCacheManager(); // Listing
Retrieves the view cache manager of the backend 12-30
$cacheManager->remove('user/show?id=12'); // The
pattern is not found, since the template is cached in the frontend
```

*Listing
12-30*

The solution is to initialize a `sfCache` object by hand, with the same settings as the frontend cache manager. Fortunately, all cache classes in symfony provide a `removePattern` method providing the same service as the view cache manager's `remove`.

For instance, if the `backend` application needs to clear the cache of the `user/show` action in the `frontend` application for the user of id 12, it can use the following:

```
$frontend_cache_dir =
sfConfig::get('sf_cache_dir').DIRECTORY_SEPARATOR.'frontend'.
```

*Listing
12-31*

```
DIRECTORY_SEPARATOR.sfConfig::get('sf_environment').DIRECTORY_SEPARATOR.'template';
$cache = new sfFileCache(array('cache_dir' => $frontend_cache_dir)); // Use the same settings as the ones defined in the frontend factories.yml
$cache->removePattern('user/show?id=12');
```

For different caching strategies, you just need to change the cache object initialization, but the cache removal process remains the same:

```
$cache = new sfMemcacheCache(array('prefix' => 'frontend'));
$cache->removePattern('user/show?id=12');
```

*Listing
12-32*

Testing and Monitoring Caching

HTML caching, if not properly handled, can create incoherence in displayed data. Each time you disable the cache for an element, you should test it thoroughly and monitor the execution boost to tweak it.

Building a 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 disabled by default in development. If you enable the HTML cache for some actions, you should add a new environment, called staging in this section, with the same settings as the prod environment (thus, with cache enabled) but with `web_debug` set to `true`.

To set it up, edit the `settings.yml` file of your application and add the lines shown in Listing 12-12 at the top.

Listing 12-12 - Settings for a staging Environment, in frontend/config/settings.yml

```
Listing 12-33
staging:
  .settings:
    web_debug: true
    cache:      true
```

In addition, create a new front controller by copying the production one (probably `myproject/web/index.php`) to a new `frontend_staging.php`. Edit it to change the arguments passed to the `getApplicationConfiguration()` method, as follows:

```
Listing 12-34
$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'staging',
true);
```

That's it—you have a new environment. Use it by adding the front controller name after the domain name:

```
Listing 12-35
http://myapp.example.com/frontend_staging.php/user/list
```

Monitoring Performance

Chapter 16 will explore the web debug toolbar and its contents. However, as this toolbar offers valuable information about cached elements, here is a brief description of its cache features.

When you browse to a page that contains cacheable elements (action, partials, fragments, and so on), the web debug toolbar (in the top-right corner of the window) shows an ignore cache button (a green, rounded arrow), as shown in Figure 12-4. This button reloads the page and forces the processing of cached elements. Be aware that it does not clear the cache.

The last number on the right side of the debug toolbar is the duration of the request execution. If you enable cache on a page, this number should decrease the second time you load the page, since Symfony uses the data from the cache instead of reprocessing the scripts. You can easily monitor the cache improvements with this indicator.

Figure 12-4 - Web debug toolbar for pages using caching



The debug toolbar also shows the number of database queries executed during the processing of the request, and the detail of the durations per category (click the total duration to display the detail). Monitoring this data, in conjunction with the total duration, will help you do fine measures of the performance improvements brought by the cache.

Benchmarking

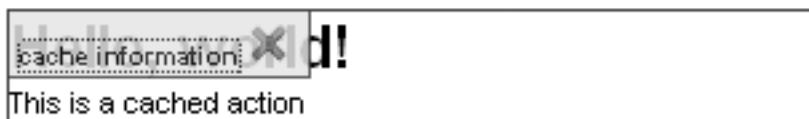
The debug mode greatly decreases the speed of your application, since a lot of information is logged and made available to the web debug toolbar. So the processed time displayed when you browse in the **staging** environment is not representative of what it will be in production, where the debug mode is turned off.

To get a better view of the process time of each request, you should use benchmarking tools, like Apache Bench or JMeter. These tools allow load testing and provide two important pieces of information: the average loading time of a specific page and the maximum capacity of your server. The average loading time data is very useful for monitoring performance improvements due to cache activation.

Identifying Cache Parts

When the web debug toolbar is enabled, the cached elements are identified in a page with a red frame, each having a cache information box on the top left, as shown in Figure 12-5. The box has a blue background if the element has been executed, or a yellow background if it comes from the cache. Clicking the cache information link displays the identifier of the cache element, its lifetime, and the elapsed time since its last modification. This will help you identify problems when dealing with out-of-context elements, to see when the element was created and which parts of a template you can actually cache.

Figure 12-5 - Identification for cached elements in a page



HTTP 1.1 and Client-Side Caching

The HTTP 1.1 protocol defines a bunch of headers that can be of great use to further speed up an application by controlling the browser's cache system.

The HTTP 1.1 specifications of the World Wide Web Consortium (W3C, <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>⁵⁰) describe these headers in detail. If an action has caching enabled, and it uses the `with_layout` option, it can use one or more of the mechanisms described in the following sections.

Even if some of the browsers of your website's users may not support HTTP 1.1, there is no risk in using the HTTP 1.1 cache features. A browser receiving headers that it doesn't understand simply ignores them, so you are advised to set up the HTTP 1.1 cache mechanisms.

In addition, HTTP 1.1 headers are also understood by proxies and caching servers. Even if a user's browser doesn't understand HTTP 1.1, there can be a proxy in the route of the request to take advantage of it.

Adding an ETag Header to Avoid Sending Unchanged Content

When the ETag feature is enabled, the web server adds to the response a special header containing a signature of the response itself.

50. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

Listing 12-36 **ETag: "1A2Z3E4R5T6Y7U"**

The user's browser will store this signature, and send it again together with the request the next time it needs the same page. If the new signature shows that the page didn't change since the first request, the server doesn't send the response back. Instead, it just sends a **304: Not modified** header. It saves CPU time (if gzipping is enabled for example) and bandwidth (page transfer) for the server, and time (page transfer) for the client. Overall, pages in a cache with an ETag are even faster to load than pages in a cache without an ETag.

In symfony, you enable the ETag feature for the whole application in `settings.yml`. Here is the default ETag setting:

Listing 12-37

```
all:
  .settings:
    etag: true
```

For actions in a cache with layout, the response is taken directly from the `cache/` directory, so the process is even faster.

Adding a Last-Modified Header to Avoid Sending Still Valid Content

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:

Listing 12-38 **Last-Modified: Sat, 23 Nov 2010 13:27:31 GMT**

Browsers can understand this header and, when requesting the page again, add an **If-Modified-Since** header accordingly:

Listing 12-39 **If-Modified-Since: Sat, 23 Nov 2010 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 as with ETAGs.

In symfony, you can set the **Last-Modified** response header just as you would for another header. For instance, you can use it like this in an action:

Listing 12-40

```
$this->getResponse()->setHttpHeader('Last-Modified',
$this->getResponse()->getDate($timestamp));
```

This date can be the actual date of the last update of the data used in the page, given from your database or your file system. The `getDate()` method of the `sfResponse` object converts a timestamp to a formatted date in the format needed for the **Last-Modified** header (RFC1123).

Adding Vary Headers to Allow Several Cached Versions of a Page

Another HTTP 1.1 header is **Vary**. It defines which parameters a page depends on, and is used by browsers and proxies to build cache keys. For example, if the content of a page depends on cookies, you can set its **Vary** header as follows:

Listing 12-41 **Vary: Cookie**

Most often, it is difficult to enable caching on actions 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, set the **Vary** header of the response properly. This can be done for the whole

application or on a per-action basis, using the `cache.yml` configuration file or the `sfResponse` related method as follows:

```
$this->getResponse()->addVaryHttpHeader('Cookie');
$this->getResponse()->addVaryHttpHeader('User-Agent');
$this->getResponse()->addVaryHttpHeader('Accept-Language');
```

*Listing
12-42*

Symfony will store a different version of the page in the cache for each value of these parameters. This will increase the size of the cache, but whenever the server receives a request matching these headers, the response is taken from the cache instead of being processed. This is a great performance tool for pages that vary only according to request headers.

Adding a Cache-Control Header to Allow Client-Side Caching

Up to now, even by adding headers, the browser keeps sending requests to the server even if it holds a cached version of the page. You can avoid that by adding `Cache-Control` and `Expires` headers to the response. These headers are disabled by default in PHP, but symfony can override this behavior to avoid unnecessary requests to your server.

As usual, you trigger this behavior by calling a method of the `sfResponse` object. In an action, define the maximum time a page should be cached (in seconds):

```
$this->getResponse()->addCacheControlHttpHeader('max_age=60');
```

*Listing
12-43*

You can also specify under which conditions a page may be cached, so that the provider's cache does not keep a copy of private data (like bank account numbers):

```
$this->getResponse()->addCacheControlHttpHeader('private=True');
```

*Listing
12-44*

Using `Cache-Control` HTTP directives, you get the ability to fine-tune the various cache mechanisms between your server and the client's browser. For a detailed review of these directives, see the W3C `Cache-Control` specifications.

One last header can be set through symfony: the `Expires` header:

```
$this->getResponse()->setHttpHeader('Expires',
$this->getResponse()->getDate($timestamp));
```

*Listing
12-45*



The major consequence of turning on the `Cache-Control` mechanism is that your server logs won't show all the requests issued by the users, but only the ones actually received. If the performance gets better, the apparent popularity of the site may decrease in the statistics.

Summary

The cache system provides variable performance boosts according to the cache type selected. From the best gain to the least, the cache types are as follows:

- Super cache
- Action cache with layout
- Action cache without layout
- Fragment cache in the template

In addition, partials and components can be cached as well.

If changing data in the model or in the session forces you to erase the cache for the sake of coherence, you can do it with a fine granularity for optimum performance—erase only the elements that have changed, and keep the others.

Remember to test all the pages where caching is enabled with extra care, as new bugs may appear if you cache the wrong elements or if you forget to clear the cache when you update the underlying data. A staging environment, dedicated to cache testing, is of great use for that purpose.

Finally, make the best of the HTTP 1.1 protocol with symfony's advanced cache-tweaking features, which will involve the client in the caching task and provide even more performance gains.

Chapter 13

I18n And L10n

If you ever developed an international application, you know that dealing with every aspect of text translation, local standards, and localized content can be a nightmare. Fortunately, symfony natively automates all the aspects of internationalization.

As it is a long word, developers often refer to internationalization as i18n (count the letters in the word to know why). Localization is referred to as l10n. They cover two different aspects of multilingual web applications.

An internationalized application contains several versions of the same content in various languages or formats. For instance, a webmail interface can offer the same service in several languages; only the interface changes.

A localized application contains distinct information according to the country from which it is browsed. Think about the contents of a news portal: When browsed from the United States, it displays the latest headlines about the United States, but when browsed from France, the headlines concern the French news. So an l10n application not only provides content translation, but the content can be different from one localized version to another.

All in all, dealing with i18n and l10n means that the application can take care of the following:

- Text translation (interface, assets, and content)
- Standards and formats (dates, amounts, numbers, and so on)
- Localized content (many versions of a given object according to a country)

This chapter covers the way symfony deals with those elements and how you can use it to develop internationalized and localized applications.

User Culture

All the built-in i18n features in symfony are based on a parameter of the user session called the culture. The culture is the combination of the country and the language of the user, and it determines how the text and culture-dependent information are displayed. Since it is serialized in the user session, the culture is persistent between pages.

Setting the Default Culture

By default, the culture of new users is the `default_culture`. You can change this setting in the `settings.yml` configuration file, as shown in Listing 13-1.

Listing 13-1 - Setting the Default Culture, in frontend/config/settings.yml

*Listing
13-1*

```
all:  
  .settings:  
    default_culture: fr_FR
```



During development, you might be surprised that a culture change in the `settings.yml` file doesn't change the current culture in the browser. That's because the session already has a culture from previous pages. If you want to see the application with the new default culture, you need to clear the domain cookies or restart your browser.

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 content for users from Spain or Mexico. The language is coded in two lowercase characters, according to the ISO 639-1 standard (for instance, `en` for English). The country is coded in two uppercase characters, according to the ISO 3166-1 standard (for instance, `GB` for Great Britain).

Changing the Culture for a User

The user culture can be changed during the browsing session—for instance, when a user decides to switch from the English version to the French version of the application, or when a user logs in to the application and uses the language stored in his preferences. That's why the `sfUser` class offers getter and setter methods for the user culture. Listing 13-2 shows how to use these methods in an action.

Listing 13-2 - Setting and Retrieving the Culture in an Action

```
Listing 13-2 // Culture setter  
$this->getUser()->setCulture('en_US');  
  
// Culture getter  
$culture = $this->getUser()->getCulture();  
=> en_US
```

Culture in the URL

When using symfony's localization and internationalization features, pages tend to have different versions for a single URL—it all depends on the user session. This prevents you from caching or indexing your pages in a search engine.

One solution is to make the culture appear in every URL, so that translated pages can be seen as different URLs to the outside world. In order to do that, add the `:sf_culture` token in every rule of your application `routing.yml`:

```
page:
    url: /:sf_culture/:page
    param: ...
    requirements: { sf_culture: (?:fr|en|de) }

article:
    url: /:sf_culture/:year/:month/:day/:slug
    param: ...
    requirements: { sf_culture: (?:fr|en|de) }
```

*Listing
13-3*

To avoid manually setting the `sf_culture` request parameter in every `link_to()`, symfony automatically adds the user culture to the default routing parameters. It also works inbound because symfony will automatically change the user culture if the `sf_culture` parameter is found in the URL.

Determining the Culture Automatically

In many applications, the user culture is defined during the first request, based on the browser preferences. Users can define a list of accepted languages in their browser, and this data is sent to the server with each request, in the `Accept-Language` HTTP header. You can retrieve it in symfony through the `sfWebRequest` object. For instance, to get the list of preferred languages of a user in an action, type this:

```
$languages = $request->getLanguages();
```

*Listing
13-4*

The HTTP header is a string, but symfony automatically parses it and converts it into an array. So the preferred language of the user is accessible with `$languages[0]` in the preceding example.

It can be useful to automatically set the user culture to its preferred browser language in a site home page or in a filter for all pages. But as your website will probably only support a limited set of languages, it's better to use the `getPreferredCulture()` method. It returns the best language by comparing the user preferred languages and the supported languages:

```
$language = $request->getPreferredCulture(array('en', 'fr'));
```

*Listing
13-5*

If there is no match, the method returns the first supported language (`en` in the preceding example).



The `Accept-Language` HTTP header is not very reliable information, since users rarely know how to modify it in their browser. Most of the time, the preferred browser language is the language of the interface, and browsers are not available in all languages. If you decide to set the culture automatically according to the browser preferred language, make sure you provide a way for the user to choose an alternate language.

Standards and Formats

The internals of a web application don't care about cultural particularities. Databases, for instance, use international standards to store dates, amounts, and so on. But when data is sent to or retrieved from users, a conversion needs to be made. Users won't understand timestamps, and they will prefer to declare their mother language as Français instead of French. So you will need assistance to do the conversion automatically, based on the user culture.

Outputting Data in the User's Culture

Once the culture is defined, the helpers depending on it will automatically have proper output. For instance, the `format_number()` helper automatically displays a number in a format familiar to the user, according to its culture, as shown in Listing 13-3.

Listing 13-3 - Displaying a Number for the User's Culture

```
Listing 13-6 <?php use_helper('Number') ?>
<?php $sf_user->setCulture('en_US') ?>
<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php $sf_user->setCulture('fr_FR') ?>
<?php echo format_number(12000.10) ?>
=> '12 000,10'
```

You don't need to explicitly pass the culture to the helpers. They will look for it themselves in the current session object. Listing 13-4 lists helpers that take into account the user culture for their output.

Listing 13-4 - Culture-Dependent Helpers

```
Listing 13-7 <?php use_helper('Date') ?>
<?php echo format_date(time()) ?>
=> '9/14/10'

<?php echo format_datetime(time()) ?>
=> 'September 14, 2010 6:11:07 PM CEST'

<?php use_helper('Number') ?>
<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php echo format_currency(1350, 'USD') ?>
=> '$1,350.00'

<?php use_helper('I18N') ?>
<?php echo format_country('US') ?>
=> 'United States'

<?php format_language('en') ?>
=> 'English'

<?php use_helper('Form') ?>
```

```
<?php echo input_date_tag('birth_date', mktime(0, 0, 0, 9, 14, 2010)) ?>
=> input type="text" name="birth_date" id="birth_date" value="9/14/10"
size="11" />

<?php echo select_country_tag('country', 'US') ?>
=> <select name="country" id="country"><option
value="AF">Afghanistan</option>
...
<option value="GB">United Kingdom</option>
<option value="US" selected="selected">United States</option>
<option value="UM">United States Minor Outlying Islands</option>
<option value="UY">Uruguay</option>
...
</select>
```

The date helpers can accept an additional format parameter to force a culture-independent display, but you shouldn't use it if your application is internationalized.

Getting Data from a Localized Input

If it is necessary to show data in the user's culture, as for retrieving data, you should, as much as possible, push users of your application to input already internationalized data. This approach will save you from trying to figure out how to convert data with varying formats and uncertain locality. For instance, who might enter a monetary value with comma separators in an input box?

You can frame the user input format either by hiding the actual data (as in a `select_country_tag()`) or by separating the different components of complex data into several simple inputs.

For dates, however, this is often not possible. Users are used to entering dates in their cultural format, and you need to be able to convert such data to an internal (and international) format. This is where the `sfI18N` class applies. Listing 13-5 demonstrates how this class is used.

Listing 13-5 - Getting a Date from a Localized Format in an Action

```
$date= $request->getParameter('birth_date');
$user_culture = $this->getUser()->getCulture();

// Getting a timestamp
$timestamp = $this->getContext()->getI18N()->getTimestampForCulture($date,
$user_culture);

// Getting a structured date
list($d, $m, $y) =
$this->getContext()->getI18N()->getDateForCulture($date, $user_culture);
```

*Listing
13-8*

Text Information in the Database

A localized application offers different content according to the user's culture. For instance, an online shop can offer products worldwide at the same price, but with a custom description for every country. This means that the database must be able to store different versions of a given piece of data, and for that, you need to design your schema in a particular way and use culture each time you manipulate localized model objects.

Creating Localized Schema

For each table that contains some localized data, you should split the table in two parts: one table that does not have any i18n column, and the other one with only the i18n columns. The two tables are to be linked by a one-to-many relationship. This setup lets you add more languages when required without changing your model. Let's consider an example using a `Product` table.

First, create tables in the `schema.yml` file, as shown in Listing 13-6.

Listing 13-6 - Sample Schema for i18n Data with Propel, in config/schema.yml

```
Listing 13-9 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, foreignReference: id }
      culture:     { isCulture: true, type: varchar, size: 7, required:
      true, primaryKey: true }
      name:         { type: varchar, size: 50 }
```

Notice the `isI18N` and `i18nTable` attributes in the first table, and the special `culture` column in the second. All these are symfony-specific Propel enhancements.

Listing 13-7 - Sample Schema for i18n Data with Doctrine, in config/doctrine/schema.yml

```
Listing 13-10 Product:
  actAs:
    I18n:
      fields: [name]
  columns:
    price: { type: float }
    name: { type: string(50) }
```

The symfony automations 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 they are related with a column named `id` in both tables, you can omit the `id` and `culture` columns in the `_i18n` table as well as the specific i18n attributes for the main table; symfony will infer them. It means that symfony will see the schema in Listing 13-8 as the same as the one in Listing 13-6.

Listing 13-8 - Sample Schema for i18n Data, Short Version, in config/schema.yml

```
Listing 13-11 my_connection:
  my_product:
    _attributes: { phpName: Product }
    id:
      price:     float
  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    name:       varchar(50)
```

Using the Generated I18n Objects

Once the corresponding object model is built (don't forget to call the `propel:build --model` task after each modification of the `schema.yml`), you can use your `Product` class with i18n support as if there were only one table, as shown in Listing 13-9.

Listing 13-9 - Dealing with i18n Objects

```
$product = ProductPeer::retrieveByPk(1);
$product->setName('Nom du produit'); // By default, the culture is the
current user culture
$product->save();

echo $product->getName();
=> 'Nom du produit'

$product->setName('Product name', 'en'); // change the value for the 'en'
culture
$product->save();

echo $product->getName('en');
=> 'Product name'
```

*Listing
13-12*

As for queries with the peer objects, you can restrict the results to objects having a translation for the current culture by using the `doSelectWithI18n` method, instead of the usual `doSelect`, as shown in Listing 13-10. In addition, it will create the related i18n objects at the same time as the regular ones, resulting in a reduced number of queries to get the full content (refer to Chapter 18 for more information about this method's positive impacts on performance).

Listing 13-10 - Retrieving Objects with an i18n Criteria

```
$c = new Criteria();
$c->add(ProductPeer::PRICE, 100, Criteria::LESS_THAN);
$products = ProductPeer::doSelectWithI18n($c, $culture);
// The $culture argument is optional
// The current user culture is used if no culture is given
```

*Listing
13-13*

So basically, you should never have to deal with the i18n objects directly, but instead pass the culture to the model (or let it guess it) each time you do a query with the regular objects.

Interface Translation

The user interface needs to be adapted for i18n applications. Templates must be able to display labels, messages, and navigation in several languages but with the same presentation. Symfony recommends that you build your templates with the default language, and that you provide a translation for the phrases used in your templates in a dictionary file. That way, you don't need to change your templates each time you modify, add, or remove a translation.

Configuring Translation

The templates are not translated by default, which means that you need to activate the template translation feature in the `settings.yml` file prior to everything else, as shown in Listing 13-11.

Listing 13-11 - Activating Interface Translation, in frontend/config/settings.yml

Listing 13-14

```
all:
  .settings:
    i18n: true
```

Using the Translation Helper

Let's say that you want to create a website in English and French, with English being the default language. Before even thinking about having the site translated, you probably wrote the templates something like the example shown in Listing 13-12.

Listing 13-12 - A Single-Language Template

Listing 13-15

```
Welcome to our website. Today's date is <?php echo format_date(date()) ?>
```

For symfony to translate the phrases of a template, they must be identified as text to be translated. This is the purpose of the `__()` helper (two underscores), a member of the I18N helper group. So all your templates need to enclose the phrases to translate in such function calls. Listing 13-11, for example, can be modified to look like Listing 13-13 (as you will see in the "Handling Complex Translation Needs" section later in this chapter, there is an even better way to call the translation helper in this example).

Listing 13-13 - A Multiple-Language-Ready Template

Listing 13-16

```
<?php use_helper('I18N') ?>

<?php echo ___('Welcome to our website.') ?>
<?php echo ___("Today's date is ") ?>
<?php echo format_date(date()) ?>
```



If your application uses the I18N helper group for every page, it is probably a good idea to include it in the `standard_helpers` setting in the `settings.yml` file, so that you avoid repeating `use_helper('I18N')` for each template.

Using Dictionary Files

Each time the `__()` function is called, symfony looks for a translation of its argument in the dictionary of the current user's culture. If it finds a corresponding phrase, the translation is sent back and displayed in the response. So the user interface translation relies on a dictionary file.

The dictionary files are written in the XML Localization Interchange File Format (XLIFF), named according to the pattern `messages.[language code].xml`, and stored in the application `i18n/` directory.

XLIFF is a standard format based on XML. As it is well known, you can use third-party translation tools to reference all text in your website and translate it. Translation firms know how to handle such files and to translate an entire site just by adding a new XLIFF translation.



In addition to the XLIFF standard, symfony also supports several other translation back-ends for dictionaries: gettext, MySQL, and SQLite. Refer to the API documentation for more information about configuring these back-ends.

Listing 13-14 shows an example of the XLIFF syntax with the `messages.fr.xml` file necessary to translate Listing 13-13 into French.

Listing 13-14 - An XLIFF Dictionary, in frontend/i18n/messages.fr.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN"
"http://www.oasis-open.org/committees/xliff/documents/xliff.dtd" >
<xliff version="1.0">
    <file original="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>
```

Listing
13-17

The `source-language` attribute must always contain the full ISO code of your default culture. Each translation is written in a `trans-unit` tag with a unique `id` attribute.

With the default user culture (set to `en_US`), the phrases are not translated and the raw arguments of the `__()` calls are displayed. The result of Listing 13-12 is then similar to Listing 13-11. However, if the culture is changed to `fr_FR` or `fr_BE`, the translations from the `messages.fr.xml` file are displayed instead, and the result looks like Listing 13-15.

Listing 13-15 - A Translated Template

```
Bienvenue sur notre site web. La date d'aujourd'hui est
<?php echo format_date(date()) ?>
```

Listing
13-18

If additional translations need to be done, simply add a new `messages.XX.xml` translation file in the same directory.



As looking for dictionary files, parsing them, and finding the correct translation for a given string takes some time, symfony uses an internal cache to speedup the process. By default, this cache uses the filesystem. You can configure how the i18N cache works (for instance, to share the cache between several servers) in the `factories.yml` (see Chapter 19).

Managing Dictionaries

If your `messages.XX.xml` file becomes too long to be readable, you can always split the translations into several dictionary files, named by theme. For instance, you can split the `messages.fr.xml` file into these three files in the application `i18n/` directory:

- `navigation.fr.xml`
- `terms_of_service.fr.xml`
- `search.fr.xml`

Note that as soon as a translation is not to be found in the default `messages.XX.xml` file, you must declare which dictionary is to be used each time you call the `__()` helper, using its third argument. For instance, to output a string that is translated in the `navigation.fr.xml` dictionary, write this:

```
<?php echo ___('Welcome to our website', null, 'navigation') ?>
```

Listing
13-19

Another way to organize translation dictionaries is to split them by module. Instead of writing a single `messages.XX.xml` file for the whole application, you can write one in each `modules/[module_name]/i18n/` directory. It makes modules more independent from the application, which is necessary if you want to reuse them, such as in plug-ins (see Chapter 17).

As updating the i18n dictionaries by hand is quite error prone, symfony provides a task to automate the process. The `i18n:extract` task parses a symfony application to extract all the strings that need to be translated. It takes an application and a culture as its arguments:

Listing 13-20 `$ php symfony i18n:extract frontend en`

By default, the task does not modify your dictionaries, it just outputs the number of new and old i18n strings. To append the new strings to your dictionary, you can pass the `--auto-save` option:

Listing 13-21 `$ php symfony i18n:extract --auto-save frontend en`

You can also delete old strings automatically by passing the `--auto-delete` option:

Listing 13-22 `$ php symfony i18n:extract --auto-save --auto-delete frontend en`



The current task has some known limitations. It can only work with the default `messages` dictionary, and for file based backends (XLIFF or gettext), it only saves and deletes strings in the main `apps/frontend/i18n/messages.XX.xml` file.

Handling Other Elements Requiring Translation

The following are other elements that may require translation:

- Images, text documents, or any other type of assets can also vary according to the user culture. The best example is a piece of text with a special typography that is actually an image. For these, you can create subdirectories named after the user culture:

Listing 13-23 `<?php echo image_tag($sf_user->getCulture().'/myText.gif') ?>`

- Error messages from validation files are automatically output by a `__()`, so you just need to add their translation to a dictionary to have them translated.
- The default symfony pages (page not found, internal server error, restricted access, and so on) are in English and must be rewritten in an i18n application. You should probably create your own `default` module in your application and use `__()` in its templates. Refer to Chapter 19 to see how to customize these pages.

Handling Complex Translation Needs

Translation only makes sense if the `__()` argument is a full sentence. However, as you sometimes have formatting or variables mixed with words, you could be tempted to cut sentences into several chunks, thus calling the helper on senseless phrases. Fortunately, the `__()` helper offers a replacement feature based on tokens, which will help you to have a meaningful dictionary that is easier to handle by translators. As with HTML formatting, you can leave it in the helper call as well. Listing 13-16 shows an example.

Listing 13-16 - Translating Sentences That Contain Code

Listing 13-24

```
// Base example
Welcome to all the <b>new</b> users.<br />
There are <?php echo count_logged() ?> persons logged.

// Bad way to enable text translation
<?php echo __('Welcome to all the') ?>
<b><?php echo __('new') ?></b>
<?php echo __('users') ?>.<br />
<?php echo __('There are') ?>
<?php echo count_logged() ?>
<?php echo __('persons logged') ?>

// Good way to enable text translation
<?php echo __('Welcome to all the <b>new</b> users') ?> <br />
<?php echo __('There are %1% persons logged', array('%1%' =>
count_logged())) ?>
```

In this example, the token is `%1%`, but it can be anything, since the replacement function used by the translation helper is `strtr()`.

One of the common problems with translation is the use of the plural form. According to the number of results, the text changes but not in the same way according to the language. For instance, the last sentence in Listing 13-16 is not correct if `count_logged()` returns 0 or 1. You could do a test on the return value of this function and choose which sentence to use accordingly, but that would represent a lot of code. Additionally, different languages have different grammar rules, and the declension rules of plural can be quite complex. As this problem is very common, symfony provides a helper to deal with it, called `format_number_choice()`. Listing 13-17 demonstrates how to use this helper.

Listing 13-17 - Translating Sentences Depending on the Value of Parameters

```
<?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()) ?>
```

*Listing
13-25*

The first argument is the multiple possibilities of text. The second argument is the replacement pattern (as with 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 (`|`) character followed by an array of acceptable values, using the following syntax:

- `[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 values greater or equal to negative infinity and strictly less than 0
- `{n: n % 10 > 1 && n % 10 < 5}` `pliki`: Matches numbers like 2, 3, 4, 22, 23, 24 (useful for languages like Polish or Russian)

Any nonempty combinations of the delimiters of square brackets and parentheses are acceptable.

The message will need to appear explicitly in the XLIFF file for the translation to work properly. Listing 13-18 shows an example.

Listing 13-18 - XLIFF Dictionary for a format_number_choice() Argument

```
...
<trans-unit id="3">
```

*Listing
13-26*

```

<source>[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf]There
are %1% persons logged</source>
<target>[0]Personne n'est connecté|[1]Une personne est
connectée|(1,+Inf]Il y a %1% personnes en ligne</target>
</trans-unit>
...

```

A few words about charsets

Dealing with internationalized content in templates often leads to problems with charsets. If you use a localized charset, you will need to change it each time the user changes culture. In addition, the templates written in a given charset will not display the characters of another charset properly.

This is why, as soon as you deal with more than one culture, all your templates must be saved in UTF-8, and the layout must declare the content with this charset. You won't have any unpleasant surprises if you always work with UTF-8, and you will save yourself from a big headache.

Symfony applications rely on one central setting for the charset, in the `settings.yml` file. Changing this parameter will change the `content-type` header of all responses.

Listing 13-27

```

all:
  .settings:
    charset: utf-8

```

Calling the Translation Helper Outside a Template

Not all the text that is displayed in a page comes from templates. That's why you often need to call the `__()` helper in other parts of your application: actions, filters, model classes, and so on. Listing 13-19 shows how to call the helper in an action by retrieving the current instance of the `I18N` object through the context singleton.

Listing 13-19 - Calling __() in an Action

Listing 13-28

```
$this->getContext()->getI18N()->__( $text, $args, 'messages' );
```

Summary

Handling internationalization and localization in web applications is painless if you know how to deal with the user culture. The helpers automatically take it into account to output correctly formatted data, and the localized content from the database is seen as if it were part of a simple table. As for the interface translation, the `__()` helper and XLIFF dictionary ensure that you will have maximum versatility with minimum work.

Chapter 14

Admin Generator

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

Code Generation Based on the Model

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

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

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

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

Example Data Model

Throughout this chapter, the listings will demonstrate the capabilities of the symfony admin generator based on a simple example, which will remind you of Chapter 8. This is the well-known example of the weblog application, containing two `BlogArticle` and `BlogComment` classes. Listing 14-1 shows its schema, illustrated in Figure 14-1.

Listing 14-1 - Propel schema of the Example Weblog Application

```
propel:  
  blog_category:  
    id: ~  
    name: varchar(255)  
  blog_author:  
    id: ~  
    name: varchar(255)
```

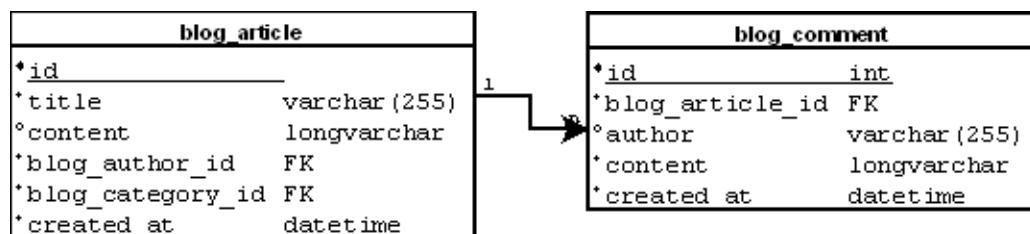
*Listing
14-1*

```

blog_article:
  id: ~
  title: varchar(255)
  content: longvarchar
  blog_author_id: ~
  blog_category_id: ~
  is_published: boolean
  created_at: ~
blog_comment:
  id: ~
  blog_article_id: ~
  author: varchar(255)
  content: longvarchar
  created_at: ~

```

Figure 14-1 - Example data model



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



To get the most out of this chapter, you need to actually do the examples. You will get a better understanding of what symfony generates and what can be done with the generated code if you have a view of every step described in the listings. Initializing the model is as simple as calling the `propel:build` task:

Listing 14-2 \$ php symfony propel:build --all --no-confirmation

As the generated admin interface relies on some magic methods to ease your task, create a `__toString()` method for each of your model class.

Listing 14-3

```

class BlogAuthor extends BaseBlogAuthor
{
    public function __toString()
    {
        return $this->getName();
    }
}

class BlogCategory extends BaseBlogCategory
{
    public function __toString()
    {
        return $this->getName();
    }
}

class BlogArticle extends BaseBlogArticle
{
    public function __toString()

```

```
{
    return $this->getTitle();
}
}
```

Administration

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

```
$ php symfony generate:app backend
```

*Listing
14-4*

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



Even if most common requirements are covered by the `generator.yml` configuration file, you can also configure an administration module via a configuration class as we will see later in this chapter.

Initiating an Administration Module

With symfony, you build an administration on a per-model basis. A module is generated based on a Propel or a Doctrine object using the `propel:generate-admin` task:

```
// Propel
$ php symfony propel:generate-admin backend BlogArticle --module=article
```

*Listing
14-5*

```
// Doctrine
$ php symfony doctrine:generate-admin backend BlogArticle --module=article
```



The administration modules are based on a REST architecture. The `propel:generate-admin` task automatically adds such a route to the `routing.yml` configuration file:

```
# apps/backend/config/routing.yml
article:
    class: sfPropelRouteCollection
    options:
        model:          BlogArticle
        module:         article
        with_wildcard_routes: true
```

*Listing
14-6*

You can also create your own route and pass the name as an argument to the task instead of the model class name:

Listing 14-7 \$ php symfony propel:generate-admin backend article --module=article

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

Listing 14-8 http://localhost/backend_dev.php/article

The look and feel of a generated module, illustrated in Figures 14-2 and 14-3, is sophisticated enough to make it usable out of the box for a commercial application.



If you don't see the expected look and feel (no stylesheet and no image), this is because you need to install the assets in your project by running the `plugin:publish-assets` task:

Listing 14-9 \$ php symfony plugin:publish-assets

Figure 14-2 - list view of the article module in the backend application

Article List							
	Id	Title	Content	Blog author	Blog category	Is published	Created at
<input type="checkbox"/>	1	Welcome to the symfony weblog!	This is the first post of this weblog.	1	1		April 1, 2010 6:30 PM
1 result							
<input type="button" value="Choose an action"/>		<input type="button" value="go"/>		<input type="button" value="New"/>			

Figure 14-3 - edit view of the article module in the backend application

Edit Article

Title	<input type="text" value="Life is beautiful"/>
Content	<input type="text" value="The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours ?"/>
Blog author	<input type="text" value="Fabien"/>
Blog category	<input type="text" value="Community"/>
Is published	<input checked="" type="checkbox"/>
Created at	<input type="text" value="04"/> / <input type="text" value="01"/> / <input type="text" value="2010"/> : <input type="text" value="16"/> : <input type="text" value="40"/>
<input type="button" value="Delete"/> <input type="button" value="Back to list"/> <input type="button" value="Save"/>	

A Look at the Generated Code

The code of the article administration module, in the `apps/backend/modules/article/` directory, looks empty because it is only initiated. The best way to review the generated code of this module is to interact with it using the browser, and then check the contents of the `cache/` folder. Listing 14-2 lists the generated actions and the templates found in the cache.

Listing 14-2 - Generated Administration Elements, in `cache/backend/ENV/modules/autoArticle/`

```
// Actions in actions/actions.class.php
index           // Displays the list of the records of the table
filter          // Updates the filters used by the list
new             // Displays the form to create a new record
create          // Creates a new record
edit            // Displays a form to modify the fields of a record
update          // Updates an existing record
delete          // Deletes a record
batch           // Executes an action on a list of selected records
batchDelete     // Executes a delete action on a list of selected records

// In templates/
_assets.php
_filters.php
```

*Listing
14-10*

```
_filters_field.php
_flashes.php
_form.php
_form_actions.php
_form_field.php
_form_fieldset.php
_form_footer.php
_form_header.php
_list.php
_list_actions.php
_list_batch_actions.php
_list_field_boolean.php
_list_footer.php
_list_header.php
_list_td_actions.php
_list_td_batch_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
_pagination.php
editSuccess.php
indexSuccess.php
newSuccess.php
```

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

Introducing the `generator.yml` Configuration File

The generated administration modules rely on parameters found in the `generator.yml` YAML configuration file. To see the default configuration of a newly created administration module, open the `generator.yml` file, located in the `backend/modules/article/config/generator.yml` directory and reproduced in Listing 14-3.

Listing 14-3 - Default Generator Configuration, in `backend/modules/article/config/generator.yml`

```
Listing 14-11 generator:
  class: sfPropelGenerator
  param:
    model_class:           BlogArticle
    theme:                 admin
    non_verbose_templates: true
    with_show:              false
    singular:               BlogArticle
    plural:                BlogArticles
    route_prefix:           blog_article
    with Propel_route:      1
    actions_base_class:     sfActions

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

```
edit: ~
new: ~
```

This configuration is enough to generate the basic administration. Any customization is added under the `config` key. Listing 14-4 shows a typical customized `generator.yml`.

Listing 14-4 - Typical Complete Generator Configuration

```
generator:
  class: sfPropelGenerator
  param:
    model_class: BlogArticle
    theme: admin
    non_verbose_templates: true
    with_show: false
    singular: BlogArticle
    plural: BlogArticles
    route_prefix: blog_article
    with_propel_route: 1
    actions_base_class: sfActions

  config:
    actions:
      _new: { label: "Create a new article" }

    fields:
      author_id: { label: Article author }
      published_on: { credentials: editor }

    list:
      title: Articles
      display: [title, blog_author, blog_category]
      fields:
        published_on: { date_format: dd/MM/yy }
      layout: stacked
      params:
        |
        %%is_published%%<strong>%%=title%%</strong><br /><em>by
        %%blog_author%%
        in %%blog_category%% (%>created_at%</em><p>%>content%</p>
      max_per_page: 2
      sort: [title, asc]

    filter:
      display: [title, blog_category_id, blog_author_id, is_published]

    form:
      display:
        "Post": [title, blog_category_id, content]
        "Workflow": [blog_author_id, is_published, created_at]
      fields:
        published_at: { help: "Date of publication" }
        title: { attributes: { style: "width: 350px" } }

    new:
      title: New article

    edit:
      title: Editing article "%%title%%"
```

*Listing
14-12*

In this configuration, there are six sections. Four of them represent views (`list`, `filter`, `new`, and `edit`) and two of them are “virtuals” (`fields` and `form`) and only exists for configuration purpose.

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

Generator Configuration

The generator configuration file is very powerful, allowing you to alter the generated administration in many ways. But such capabilities come with a price: The overall syntax description is long to read and learn, making this chapter one of the longest in this book.

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

Listing 14-13 \$ php symfony propel:generate-admin backend BlogComment --module=comment

Figure 14-4 - The administration generator cheat sheet

Fields

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

Field Settings

The administration generator creates a `field` for each column in the `schema.yml` file. Under the `fields` key, you can modify the way each field is displayed, formatted, etc. For

instance, the field settings shown in Listing 14-5 define a custom label class and input type for the `title` field, and a label and a tooltip for the `content` field. The following sections will describe in detail how each parameter works.

Listing 14-5 - Setting a Custom Label for a Column

```
config:
  fields:
    title:
      label: Article Title
      attributes: { class: foo }
      content: { label: Body, help: Fill in the article body }
```

*Listing
14-14*

In addition to this default definition for all the views, you can override the field settings for a given view (`list`, `filter`, `form`, `new`, and `edit`), as demonstrated in Listing 14-6.

Listing 14-6 - Overriding Global Settings View per View

```
config:
  fields:
    title: { label: Article Title }
    content: { label: Body }

  list:
    fields:
      title: { label: Title }

  form:
    fields:
      content: { label: Body of the article }
```

*Listing
14-15*

This is a general principle: Any settings that are set for the whole module under the `fields` key can be overridden by view-specific areas. The overriding rules are the following:

- `new` and `edit` inherits from `form` which inherits from `fields`
- `list` inherits from `fields`
- `filter` inherits from `fields`

Adding Fields to the Display

The fields that you define in the `fields` section can be displayed, hidden, ordered, and grouped in various ways for each view. The `display` key is used for that purpose. For instance, to arrange the fields of the `comment` module, use the code of Listing 14-7.

Listing 14-7 - Choosing the Fields to Display, in modules/comment/config/generator.yml

```
config:
  fields:
    article_id: { label: Article }
    created_at: { label: Published on }
    content: { label: Body }

  list:
    display: [id, blog_article_id, content]

  form:
    display:
```

*Listing
14-16*

```
NONE:      [blog_article_id]
Editable: [author, content, created_at]
```

The list will then display three columns, as in Figure 14-5, and the new and edit form will display four fields, assembled in two groups, as in Figure 14-6.

Figure 14-5 - Custom column setting in the list view of the comment module

<input type="checkbox"/>	Id	Blog article	Body	Actions
<input type="checkbox"/>	1	1	First !	Edit Delete
<input type="checkbox"/>	2	1	Oh too bad i wanted to be first !	Edit Delete
<input type="checkbox"/>	3	1	This is a really nice blog.	Edit Delete
<input type="checkbox"/>	4	2	Great story !	Edit Delete
<input type="checkbox"/>	5	2	Hello Fabien	Edit Delete

5 results

Figure 14-6 - Grouping fields in the edit view of the comment module

Edit Comment

Blog article	<input type="text" value="Welcome to the symfony weblog!"/>
Editable	
Author	<input type="text" value="Benoit"/>
Body	<input type="text" value="This is a really nice blog."/>
Published on	<input type="text" value="04"/> / <input type="text" value="01"/> / <input type="text" value="2010"/> <input type="text" value="17"/> : <input type="text" value="10"/>
<input alt="Delete icon"/> <input type="button" value="Delete"/> <input alt="Back to list icon"/> <input type="button" value="Back to list"/> <input type="button" value="Save"/>	

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

- For the `list` view: Put the fields in a simple array to select the columns to display and the order in which they appear.
- For the `form`, `new`, and `edit` views: Use an associative array to group fields with the group name as a key, or `NONE` for a group with no name. The value is still an array of ordered column names. Be careful to list all the required fields referenced in your form class or you may have some unexpected validation errors (see Chapter 10).

Custom Fields

As a matter of fact, the fields configured in `generator.yml` don't even need to correspond to actual columns defined in the schema. If the related class offers a custom getter, it can be used as a field for the `list` view; if there is a getter and/or a setter, it can also be used in the `edit` view. For instance, you can extend the `BlogArticle` model with a `getNbComments()` method similar to the one in Listing 14-8.

Listing 14-8 - Adding a Custom Getter in the Model, in lib/model/BlogArticle.php

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

*Listing
14-17*

Then `nb_comments` is available as a field in the generated module (notice that the getter uses a camelCase version of the field name), as in Listing 14-9.

Listing 14-9 - Custom Getters Provide Additional Columns for Administration Modules, in backend/modules/article/config/generator.yml

```
config:
  list:
    display: [title, blog_author, blog_category, nb_comments]
```

*Listing
14-18*

The resulting `list` view of the `article` module is shown in Figure 14-07.

Figure 14-07 - Custom field in the list view of the article module

<input type="checkbox"/> Title	Blog author	Blog category	Nb comments	Actions
<input type="checkbox"/> Welcome to the symfony weblog!	1	1	3	Edit Delete
<input type="checkbox"/> Life is beautiful	1	1	2	Edit Delete

2 results

Choose an action Create a new article

Partial Fields

The code located in the model must be independent from the presentation. The example of the `getArticleLink()` method earlier doesn't respect this principle of layer separation, because some view code appears in the model layer. As a matter of fact, if you try to use this configuration, you will end up with the link being displayed as an `<a>` tag as it is escaped by default. To achieve the same goal in a correct way, you'd better put the code that outputs HTML for a custom field in a partial. Fortunately, the administration generator allows it if you declare a field name prefixed by an underscore. In that case, the `generator.yml` file of Listing 14-11 is to be modified as in Listing 14-12.

Listing 14-12 - Partial Can Be Used As Additional Columns—Use the _ Prefix

Listing 14-19

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

For this to work, an `_article_link.php` partial must be created in the `modules/comment/templates/` directory, as in Listing 14-13.

Listing 14-13 - Example Partial for the list View, in modules/comment/templates/_article_link.php

Listing 14-20

```
<?php echo link_to($BlogComment->getBlogArticle()->getTitle(),
  'blog_article_edit', $BlogComment->getBlogArticle()) ?>
```

Notice that the partial template of a partial field has access to the current object through a variable named by the class (`$BlogComment` in this example).

Figure 14-08 - Partial field in the list view of the article module

<input type="checkbox"/>	Id	Article link	Body	Actions
<input type="checkbox"/>	1	Welcome to the symfony weblog!	First !	Edit Delete
<input type="checkbox"/>	2	Welcome to the symfony weblog!	Oh too bad i wanted to be first !	Edit Delete
<input type="checkbox"/>	3	Welcome to the symfony weblog!	This is a really nice blog.	Edit Delete
<input type="checkbox"/>	4	Life is beautiful	Great story !	Edit Delete
<input type="checkbox"/>	5	Life is beautiful	Hello Fabien	Edit Delete

5 results

New

The layer separation is respected. If you get used to respecting the layer separation, you will end up with more maintainable applications.

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

Listing 14-14 - Partial Field Properties Can Be Customized Under the fields Key

Listing 14-21

```
config:
  fields:
    article_link: { label: Article }
```

If your partial becomes crowded with logic, you'll probably want to replace it with a component. Change the `_` prefix to `~` and you can define a component field, as you can see in Listing 14-15.

Listing 14-15 - Components Can Be Used As Additional Columns—Use the ~ Prefix

Listing 14-22

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

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



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

View Customization

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

Changing the View Title

In addition to a custom set of fields, the `list`, `new`, and `edit` pages can have a custom page title. For instance, if you want to customize the title of the `article` views, do as in Listing 14-16. The resulting `edit` view is illustrated in Figure 14-09.

Listing 14-16 - Setting a Custom Title for Each View, in backend/modules/article/config/generator.yml

```
config:  
  list:  
    title: List of Articles  
  
  new:  
    title: New Article  
  
  edit:  
    title: Edit Article %title% (%id%)
```

*Listing
14-23*

Figure 14-09 - Custom title in the edit view of the article module

Editing article "Welcome to the symfony weblog!"

Post	
Title	Welcome to the symfony weblog!
Blog category	Community ▾
Content	This is the first post of this weblog.
Workflow	
Blog author	Fabien ▾
Is published	<input checked="" type="checkbox"/>
Created at	04 ▾ / 01 ▾ / 2010 ▾ 18 ▾ : 30 ▾
 Delete Back to list Save	

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



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

Adding Tooltips

In the `list`, `new`, `edit`, and `filter` views, you can add tooltips to help describe the fields that are displayed. For instance, to add a tooltip to the `blog_article_id` field of the `edit` view of the `comment` module, add a `help` property in the `fields` definition as in Listing 14-17. The result is shown in Figure 14-10.

Listing 14-17 - Setting a Tooltip in the edit View, in modules/comment/config/generator.yml

```
Listing 14-24 config:
  edit:
    fields:
      blog_article_id: { help: The current comment relates to this article }
```

Figure 14-10 - Tooltip in the edit view of the comment module

Edit Comment

Blog article

Welcome to the symfony weblog! ▾

The current comment relates to this article

In the `list` view, tooltips are displayed in the column header; in the `new`, `edit`, and `filter` views, they appear under the field tag.

Modifying the Date Format

Dates can be displayed using a custom format as soon as you use the `date_format` option, as demonstrated in Listing 14-18.

Listing 14-18 - Formatting a Date in the list View

```
config:  
  list:  
    fields:  
      created_at: { label: Published, date_format: dd/MM }
```

*Listing
14-25*

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

Administration templates are i18N ready

The admin generated modules are made of interface strings (default action names, pagination help messages, ...) and custom strings (titles, column labels, help messages, error messages, ...).

Translations of the interface strings are bundled with symfony for a lot of languages. But you can also add your own or override existing ones by creating a custom XLIFF file in your `i18n` directory for the `sf_admin` catalogue (`apps/frontend/i18n/sf_admin.XX.xml` where `XX` is the ISO code of the language).

All the custom strings found in the generated templates are also automatically internationalized (i.e., enclosed in a call to the `__()` helper). This means that you can easily translate a generated administration by adding the translations of the phrases in an XLIFF file, in your `apps/frontend/i18n/` directory, as explained in the previous chapter.

You can change the default catalogue used for the custom strings by specifying an `i18n_catalogue` parameter:

```
generator:  
  class: sfPropelGenerator  
  param:  
    i18n_catalogue: admin
```

*Listing
14-26*

List View-Specific Customization

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



Changing the Layout

By default, the hyperlink between the `list` view and the `edit` view is borne by the primary key column. If you refer back to Figure 14-08, you will see that the `id` column in the comment list not only shows the primary key of each comment, but also provides a hyperlink allowing users to access the `edit` view.

If you prefer the hyperlink to the detail of the record to appear on another column, prefix the column name by an equal sign (=) in the `display` key. Listing 14-19 shows how to remove the `id` from the displayed fields of the comment `list` and to put the hyperlink on the `content` field instead. Check Figure 14-11 for a screenshot.

Listing 14-19 - Moving the Hyperlink for the edit View in the list View, in modules/comment/config/generator.yml

```
Listing 14-27 config:
list:
    display: [_article_link, =content]
```

Figure 14-11 - Moving the link to the edit view on another column, in the list view of the comment module

Article link	Body	Actions	
<input type="checkbox"/> Welcome to the symfony weblog!	First !	 Edit	 Delete
<input type="checkbox"/> Welcome to the symfony weblog!	Oh too bad i wanted to be first !	 Edit	 Delete
<input type="checkbox"/> Welcome to the symfony weblog!	This is a really nice blog.	 Edit	 Delete
<input type="checkbox"/> Life is beautiful	Great story !	 Edit	 Delete
<input type="checkbox"/> Life is beautiful	Hello Fabien	 Edit	 Delete
5 results			
<input type="button" value="Choose an action"/> <input type="button" value="go"/>		 New	

By default, the `list` view uses the `tabular` layout, where the fields appear as columns, as shown previously. But you can also use the `stacked` layout and concatenate the fields into a single string that expands on the full length of the table. If you choose the `stacked` layout, you must set in the `params` key the pattern defining the value of each line of the list. For instance, Listing 14-20 defines a stacked layout for the list view of the comment module. The result appears in Figure 14-12.

Listing 14-20 - Using a stacked Layout in the list View, in modules/comment/config/generator.yml

```
Listing 14-28 config:
list:
    layout: stacked
    params: |
        %%=content%%<br />
        (sent by %%author%% on %%created_at%% about %%_article_link%%)
    display: [created_at, author, content]
```

Figure 14-12 - Stacked layout in the list view of the comment module

Published on	Author	Body	Actions
<input type="checkbox"/> First ! (sent by Gilles on April 1, 2010 5:07 PM about Welcome to the symfony weblog!)			Edit Delete
<input type="checkbox"/> I really love playing with the symfony framework. (sent by Sylvain on April 1, 2010 5:08 PM about Welcome to the symfony weblog!)			Edit Delete
<input type="checkbox"/> This is a really nice blog. (sent by Benoît on April 1, 2010 5:10 PM about Welcome to the symfony weblog!)			Edit Delete
<input type="checkbox"/> Great story ! (sent by Matthieu on April 1, 2010 5:10 PM about Life is beautiful)			Edit Delete
<input type="checkbox"/> Hello Fabien (sent by Régis on April 1, 2010 5:11 PM about Life is beautiful)			Edit Delete

6 results (page 1/2)

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

Filtering the Results

In a `list` view, you can add a set of filter interactions. With these filters, users can both display fewer results and get to the ones they want faster. Configure the filters under the `filter` key, with an array of field names. For instance, add a filter on the `blog_article_id`, `author`, and `created_at` fields to the comment `list` view, as in Listing 14-21, to display a filter box similar to the one in Figure 14-13.

Listing 14-21 - Setting the Filters in the list View, in modules/comment/config/generator.yml

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

    filter:
      display: [blog_article_id, author, created_at]
```

Listing 14-29

Figure 14-13 - Filters in the list view of the comment module

The screenshot shows the 'Comment List' page of the Symfony Admin Generator. On the left, there is a table with columns: Published on, Author, Body, and Actions (Edit and Delete). The table contains five rows of comments from different users. On the right, there is a sidebar with three filter fields: 'Blog article' (a dropdown menu), 'Author' (a dropdown menu with an 'is empty' checkbox), and 'Published on' (two dropdown menus for 'from' and 'to' dates, each with an 'is empty' checkbox). Below the sidebar are 'Reset' and 'Filter' buttons. At the bottom of the page, there is a 'Choose an action' dropdown, a 'go' button, and a 'New' button.

The filters displayed by symfony depend on the column type defined in the schema, and can be customized in the filter form class:

- For text columns (like the `author` field in the `comment` module), the filter is a text input allowing text-based search (wildcards are automatically added).
- For foreign keys (like the `blog_article_id` field in the `comment` module), the filter is a drop-down list of the records of the related table. By default, the options of the drop-down list are the ones returned by the `_toString()` method of the related class.
- For date columns (like the `created_at` field in the `comment` module), the filter is a pair of rich date tags, allowing the selection of a time interval.
- For Boolean columns, the filter is a drop-down list having `true`, `false`, and `true or false` options—the last value reinitializes the filter.

Just like the `new` and `edit` views are tied to a form class, the filters use the default filter form class associated with the model (`BlogArticleFormFilter` for the `BlogArticle` model for example). By defining a custom class for the filter form, you can customize the filter fields by leveraging the power of the form framework and by using all the available filter widgets. It is as easy as defining a `class` under the `filter` entry as shown in Listing 14-22.

Listing 14-22 - Customizing the Form Class used for Filtering

```
Listing 14-30 config:
  filter:
    class: BackendArticleFormFilter
```



To disable filters altogether, you can just specify `false` as the `class` to use for the filters.

You can also use partial filters to implement custom filter logic. Each partial receives the `form` and the HTML `attributes` to use when rendering the form element. Listing 14-23 shows an example implementation that mimics the default behavior but with a partial.

Listing 14-23 - Using a Partial Filter

```
Listing 14-31 // Define the partial, in templates/_state.php
<?php echo $form[$name]->render($attributes->getRawValue()) ?>

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

Sorting the List

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

Figure 14-14 - Table headers of the list view are sort controls



The screenshot shows a table titled 'Articles' with two rows of data. The columns are: Title, Blog author, Blog category, Nb comments, and Actions. The 'Title' column header is underlined and has a cursor icon pointing at it, indicating it's a sort control. The first row contains 'Welcome to the symfony weblog!' with blog author '1', category '1', and 3 comments. The second row contains 'Life is beautiful' with blog author '1', category '1', and 2 comments. Each row has 'Edit' and 'Delete' buttons in the 'Actions' column. Below the table, a message says '2 results'. At the bottom, there are buttons for 'Choose an action' and 'go', and a link to 'Create a new article'.

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

```
<?php echo link_to('Comment list by date',
  '@blog_comment?sort=created_at&sort_type=desc' ) ?>
```

Listing 14-32

You can also define a default `sort` order for the `list` view directly in the `generator.yml` file. The syntax follows the example given in Listing 14-24.

Listing 14-24 - Setting a Default Sort Field in the list View

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

Listing 14-33



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

Customizing the Pagination

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

Figure 14-15 - Pagination controls appear on long lists

<input type="checkbox"/> Published on	Author	Body	Actions
<input type="checkbox"/>	First ! (sent by Gilles on April 1, 2010 5:07 PM about Welcome to the symfony weblog!)		Edit Delete
<input type="checkbox"/>	Oh too bad i wanted to be first ! (sent by Sylvain on April 1, 2010 5:08 PM about Welcome to the symfony weblog!)		Edit Delete
<input type="checkbox"/>	This is a really nice blog. (sent by Benoit on April 1, 2010 5:10 PM about Welcome to the symfony weblog!)		Edit Delete
<input type="checkbox"/>	Great story ! (sent by Matthieu on April 1, 2010 5:10 PM about Life is beautiful)		Edit Delete
<input type="checkbox"/>	Hello Fabien (sent by Régis on April 1, 2010 5:11 PM about Life is beautiful)		Edit Delete
6 results (page 1/2)			
<input type="button" value="Choose an action"/> <input type="button" value="go"/> New			

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

```
Listing 14-34 config:
list:
  max_per_page: 5
```

Using a Join to Speed Up Page Delivery

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

```
Listing 14-35 config:
list:
  peer_method: doSelectJoinBlogArticle
```

Chapter 18 explains the concept of Join more extensively.

New and Edit View-Specific Customization

In a new or edit view, the user can modify the value of each column for a new record or a given record. By default, the form used by the admin generator is the form associated with the model: `BlogArticleForm` for the `BlogArticle` model. You can customize the class to use by defining the `class` under the `form` entry as shown in Listing 14-25.

Listing 14-25 - Customizing the Form Class used for the new and edit views

```
Listing 14-36 config:
form:
  class: BackendBlogArticleForm
```

Using a custom form class allows the customization of all the widgets and validators used for the admin generator. The default form class can then be used and customized specifically for the frontend application.

You can also customize the labels, help messages, and the layout of the form directly in the `generator.yml` configuration file as shown in Listing 14-26.

Listing 14-26 - Customizing the Form Display

```
config:
  form:
    display:
      NONE: [article_id]
      Editable: [author, content, created_at]
    fields:
      content: { label: body, help: "The content can be in the Markdown
format" }
```

*Listing
14-37*

Handling Partial Fields

Partial fields can be used in the `new` and `edit` views just like in `list` views.

Dealing with Foreign Keys

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

One-to-Many Relationships

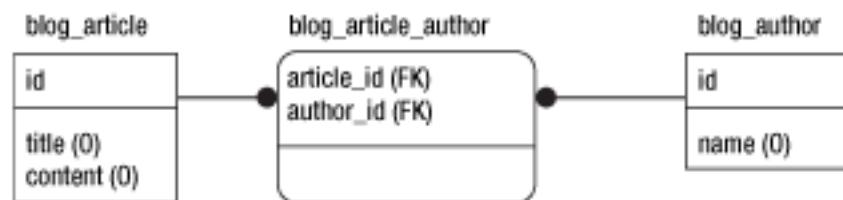
The 1-n table relationships are taken care of by the administration generator. As is depicted by Figure 14-1 earlier, the `blog_comment` table is related to the `blog_article` table through the `blog_article_id` field. If you initiate the module of the `BlogComment` class with the administration generator,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

The same goes if you need to display the list of comments related to an article in the `article` module (n-1 relationship).

Many-to-Many Relationships

Symfony also takes care of n-n table relationships out of the box, as shown in Figure 14-16.

Figure 14-16 - Many-to-many relationships



By customizing the widget used to render the relationship, you can tweak the rendering of the field (illustrated in Figure 14-17):

Figure 14-17 - Available controls for many-to-many relationships



Adding Interactions

Administration modules allow users to perform the usual CRUD operations, but you can also add your own interactions or restrict the possible interactions for a view. For instance, the interaction definition shown in Listing 14-27 gives access to all the default CRUD actions on the article module.

Listing 14-27 - Defining Interactions for Each View, in backend/modules/article/config/generator.yml

```
Listing 14-38 config:
  list:
    title:          List of Articles
    object_actions:
      _edit:        ~
      _delete:      ~
    batch_actions:
      _delete:      ~
    actions:
      _new:         ~

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

In a `list` view, there are three action settings: the actions available for every object (`object_actions`), the actions available for a selection of objects (`batch_actions`), and actions available for the whole page (`actions`). The list interactions defined in Listing 14-27 render like in Figure 14-18. Each line shows one button to edit the record and one to delete it, plus one checkbox on each line to delete a selection of records. At the bottom of the list, a button allows the creation of a new record.

Figure 14-18 - Interactions in the list view

<input type="checkbox"/>	Id	Blog article	Body	Actions	
<input type="checkbox"/>	1	1	First !	 Edit	 Delete
<input type="checkbox"/>	2	1	Oh too bad i wanted to be first !	 Edit	 Delete
<input type="checkbox"/>	3	1	This is a really nice blog.	 Edit	 Delete
<input type="checkbox"/>	4	2	Great story !	 Edit	 Delete
<input type="checkbox"/>	5	2	Hello Fabien	 Edit	 Delete
5 results					
<input data-bbox="192 765 472 804" type="button" value="Choose an action"/> <input data-bbox="488 765 541 804" type="button" value="go"/>			 New		

In a new and edit views, as there is only one record edited at a time, there is only one set of actions to define (under actions). The edit interactions defined in Listing 14-27 render like in Figure 14-23. Both the save and the save_and_add actions save the current edits in the records, the difference being that the save action displays the edit view on the current record after saving, while the save_and_add action displays a new view to add another record. The save_and_add action is a shortcut that you will find very useful when adding many records in rapid succession. As for the position of the delete action, it is separated from the other buttons so that users don't click it by mistake.

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

Figure 14-19 - Interactions in the edit view

Body of article Life is beautiful

Content:	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?
<input data-bbox="763 1581 837 1619" type="button" value="list"/> <input checked="" data-bbox="853 1581 928 1619" type="button" value="save"/> <input checked="" data-bbox="944 1581 1177 1619" type="button" value="save and add"/>	
<input checked="" data-bbox="219 1648 334 1686" type="button" value="delete"/>	

But you can also add a custom interaction, in which case you must specify a name starting with no underscore, and a target action in the current module, as in Listing 14-28.

Listing 14-28 - Defining a Custom Interaction

```
list:
  title:          List of Articles
  object_actions:
    _edit:      -
```

Listing
14-39

```
_delete: -
addcomment: { label: Add a comment, action: addComment }
```

Each article in the list will now show the `Add a comment` link, as shown in Figure 14-20. Clicking it triggers a call to the `addComment` action in the current module. The primary key of the current object is automatically added to the request parameters.

Figure 14-20 - Custom interaction in the list view

<input type="checkbox"/> Title	Nb comments	Actions
<input type="checkbox"/> Life is beautiful	2	Edit Delete Add a comment
<input type="checkbox"/> Welcome to the symfony weblog!	4	Edit Delete Add a comment
2 results		
<input type="button" value="Choose an action"/> <input type="button" value="go"/>		Create a new article

The `addComment` action can be implemented as in Listing 14-29.

Listing 14-29 - Implementing the Custom Interaction Action, in actions/actions.class.php

```
Listing 14-40
public function executeAddComment($request)
{
    $comment = new BlogComment();
    $comment->setArticleId($request->getParameter('id'));
    $comment->save();

    $this->redirect('blog_comment_edit', $comment);
}
```

Batch actions receive an array of the primary keys of the selected records in the `ids` request parameter.

One last word about actions: If you want to suppress completely the actions for one category, use an empty list, as in Listing 14-30.

Listing 14-30 - Removing All Actions in the list View

```
Listing 14-41
config:
list:
    title: List of Articles
    actions: {}
```

Form Validation

The validation is taken care of by the form used by the `new` and `edit` views automatically. You can customize it by editing the corresponding form classes.

Restricting User Actions Using Credentials

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

The fields in the generator can take a `credentials` parameter into account so as to appear only to users who have the proper credential. This works for the `list` entry. Additionally, the

generator can also hide interactions according to credentials. Listing 14-31 demonstrates these features.

Listing 14-31 - Using Credentials in generator.yml

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

  # The addcomment interaction is restricted to the users with the admin
  # credential
  actions:
    addcomment: { credentials: [admin] }
```

*Listing
14-42*

Modifying the Presentation of Generated Modules

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

Using a Custom Style Sheet

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

You can define an alternative CSS to be used for an administration module instead of a default one by adding a `css` parameter to the generator configuration, as in Listing 14-32.

Listing 14-32 - Using a Custom Style Sheet Instead of the Default One

```
generator:
  class: sfPropelGenerator
  param:
    model_class:          BlogArticle
    theme:                admin
    non_verbose_templates: true
    with_show:             false
    singular:              BlogArticle
    plural:                BlogArticles
    route_prefix:          blog_article
    with Propel_route:     1
    actions_base_class:    sfActions
    css:                  mystylesheet
```

*Listing
14-43*

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

Creating a Custom Header and Footer

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

Listing 14-44

```
_list_header.php
_list_footer.php
_form_header.php
_form_footer.php
```

For instance, if you want to add a custom header to the `article/edit` view, create a file called `_form_header.php` as in Listing 14-33. It will work with no further configuration.

Listing 14-33 - Example edit Header Partial, in modules/article/templates/_form_header.php

Listing 14-45

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

Notice that an edit partial always has access to the current object through a variable named after the class, and that a `list` partial always has access to the current pager through the `$pager` variable.

Customizing the Theme

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

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

However, if you want to override those for several modules in the same way, you should probably create a reusable theme. A theme is a sub-set of templates and actions that can be used by an administration module if specified in the `theme` value at the beginning of `generator.yml`. With the default theme, `symfony` uses the files defined in `sfConfig::get('sf_symfony_lib_dir')/plugins/sfPropelPlugin/data/generator/sfPropelModule/admin/`.

The theme files must be located in a project tree structure, in a `data/generator/sfPropelModule/[theme_name]/template/templates/` directory, and you can bootstrap a new theme by copying the files you want to override from the default theme (located in `sfConfig::get('sf_symfony_lib_dir')/plugins/sfPropelPlugin/data/generator/sfPropelModule/admin/` directory):

Listing 14-46

```
// Partials, in [theme_name]/template/templates/
_assets.php
_filters.php
_filters_field.php
_flashes.php
_form.php
_form_actions.php
_form_field.php
_form_fieldset.php
_form_footer.php
_form_header.php
_list.php
_list_actions.php
_list_batch_actions.php
_list_field_boolean.php
_list_footer.php
_list_header.php
_list_td_actions.php
```

```

_list_td_batch_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
_pagination.php
editSuccess.php
indexSuccess.php
newSuccess.php

// Actions, in [theme_name]/parts
actionsConfiguration.php
batchAction.php
configuration.php
createAction.php
deleteAction.php
editAction.php
fieldsConfiguration.php
filterAction.php
filtersAction.php
filtersConfiguration.php
indexAction.php
newAction.php
paginationAction.php
paginationConfiguration.php
processFormAction.php
sortingAction.php
sortingConfiguration.php
updateAction.php

```

Be aware that the template files are actually templates of templates, that is, PHP files that will be parsed by a special utility to generate templates based on generator settings (this is called the compilation phase). The generated templates must still contain PHP code to be executed during actual browsing, so the templates of templates use an alternative syntax to keep PHP code unexecuted for the first pass. Listing 14-34 shows an extract of a default template of template.

Listing 14-34 - Syntax of Templates of Templates

```
<h1>[?php echo <?php echo $this->getI18NString('edit.title') ?> ?]</h1>
[?php include_partial('<?php echo $this->getModuleName() ?>/flashes') ?]
```

*Listing
14-47*

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

```
<h1><?php echo __('List of all Articles') ?></h1>
<?php include_partial('article/flashes') ?>
```

*Listing
14-48*

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



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

Building Your Own Generator

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

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

Listing 14-49

```
$manager = new sfGeneratorManager();
$data = $manager->generate('sfPropelGenerator', $parameters);
```

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

Summary

To automatically generate your back-end applications, the basis is a well-defined schema and object model. The customization of the administration-generated modules are to be done mostly through configuration.

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

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

Chapter 15

Unit And Functional Testing

Automated tests are one of the greatest advances in programming since object orientation. Particularly conducive to developing web applications, they can guarantee the quality of an application even if releases are numerous. Symfony provides a variety of tools for facilitating automated testing, and these are introduced in this chapter.

Automated Tests

Any developer with experience developing web applications is well aware of the time it takes to do testing well. Writing test cases, running them, and analyzing the results is a tedious job. In addition, the requirements of web applications tend to change constantly, which leads to an ongoing stream of releases and a continuing need for code refactoring. In this context, new errors are likely to regularly crop up.

That's why automated tests are a suggested, if not required, part of a successful development environment. A set of test cases can guarantee that an application actually does what it is supposed to do. Even if the internals are often reworked, the automated tests prevent accidental regressions. Additionally, they compel developers to write tests in a standardized, rigid format capable of being understood by a testing framework.

Automated tests can sometimes replace developer documentation since they can clearly illustrate what an application is supposed to do. A good test suite shows what output should be expected for a set of test inputs, and that is a good way to explain the purpose of a method.

The symfony framework applies this principle to itself. The internals of the framework are validated by automated tests. These unit and functional tests are not bundled with the PEAR package, but you can check them out from the SVN repository or browse them online⁵¹.

Unit and Functional Tests

Unit tests confirm that a unitary code component provides the correct output for a given input. They validate how functions and methods work in every particular case. Unit tests deal with one case at a time, so for instance a single method may need several unit tests if it works differently in certain situations.

Functional tests validate not a simple input-to-output conversion, but a complete feature. For instance, a cache system can only be validated by a functional test, because it involves more than one step: The first time a page is requested, it is rendered; the second time, it is taken from the cache. So functional tests validate a process and require a scenario. In symfony, you should write functional tests for all your actions.

51. <http://trac.symfony-project.org/browser/branches/1.4/test>

For the most complex interactions, these two types may fall short. Ajax interactions, for instance, require a web browser to execute JavaScript, so automatically testing them requires a special third-party tool. Furthermore, visual effects can only be validated by a human.

If you have an extensive approach to automated testing, you will probably need to use a combination of all these methods. As a guideline, remember to keep tests simple and readable.



Automated tests work by comparing a result with an expected output. In other words, they evaluate assertions (expressions like `$a == 2`). The value of an assertion is either `true` or `false`, and it determines whether a test passes or fails. The word “assertion” is commonly used when dealing with automated testing techniques.

Test-Driven Development

In the test-driven development (TDD) methodology, the tests are written before the code. Writing tests first helps you to focus on the tasks a function should accomplish before actually developing it. It's a good practice that other methodologies, like Extreme Programming (XP), recommend as well. Plus it takes into account the undeniable fact that if you don't write unit tests first, you never write them.

For instance, imagine that you must develop a text-stripping function. The function removes white spaces at the beginning and at the end of the string, replaces non-alphabetical characters by underscores, and transforms all uppercase characters to lowercase ones. In test-driven development, you would first think about all the possible cases and provide an example input and expected output for each, as shown in Table 15-1.

Table 15-1 - A List of Test Cases for a Text-Stripping Function

Input	Expected Output
" foo "	"foo"
"foo bar"	"foo_bar"
"-)foo:..=bar?"	"_foo____bar_"
"Foobar"	"foobar"
"Don't foo-bar me!"	"don_t_foo_bar_me_"

You would write the unit tests, run them, and see that they fail. You would then add the necessary code to handle the first test case, run the tests again, see that the first one passes, and go on like that. Eventually, when all the test cases pass, the function is correct.

An application built with a test-driven methodology ends up with roughly as much test code as actual code. As you don't want to spend time debugging your tests cases, keep them simple.



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 automated tests before deploying a new release of an application in production—this is called regression testing.

The Lime Testing Framework

There are many unit test frameworks in the PHP world, with the most well known being PHPUnit. Symfony has its own, called lime. It is based on the `Test::More` Perl library, and is TAP compliant, which means that the result of tests is displayed as specified in the Test Anything Protocol, designed for better readability of test output.

Lime provides support for unit testing. It is more lightweight than other PHP testing frameworks and has several advantages:

- It launches test files in a sandbox to avoid strange side effects between each test run. Not all testing frameworks guarantee a clean environment for each test.
- Lime tests are very readable, and so is the test output. On compatible systems, lime uses color output in a smart way to distinguish important information.
- Symfony itself uses lime tests for regression testing, so many examples of unit and functional tests can be found in the symfony source code.
- The lime core is validated by unit tests.
- It is written in PHP, and it is fast, well coded, and has no dependence.

The various tests described next use the lime syntax. They work out of the box with any symfony installation.



Unit and functional tests are not supposed to be launched in production. They are developer tools, and as such, they should be run in the developer's computer, not in the host server.

Unit Tests

Symfony unit tests are simple PHP files ending in `Test.php` and located in the `test/unit/` directory of your application. They follow a simple and readable syntax.

What Do Unit Tests Look Like?

Listing 15-1 shows a typical set of unit tests for the `strtolower()` function. It starts by an instantiation of the `lime_test` object (you don't need to worry about the parameters for now). Each unit test is a call to a method of the `lime_test` instance. The last parameter of these methods is always an optional string that serves as the output.

Listing 15-1 - Example Unit Test File, in `test/unit/strtolowerTest.php`

```
<?php  
  
include dirname(__FILE__).'./..bootstrap/unit.php';  
  
$t = new lime_test(7);  
  
// strtolower()  
$t->diag('strtolower()');  
$t->isa_ok(strtolower('Foo'), 'string',  
    'strtolower() returns a string');  
$t->is(strtolower('FOO'), 'foo',  
    'strtolower() transforms the input to lowercase');  
$t->is(strtolower('foo'), 'foo',  
    'strtolower() leaves lowercase characters unchanged');  
$t->is(strtolower('12#@~'), '12#@~',  
    'strtolower() leaves non alphabetical characters unchanged');  
$t->is(strtolower('FOO BAR'), 'foo bar',  
    'strtolower() leaves blanks alone');  
$t->is(strtolower('Fo0 bAr'), 'foo bar',  
    'strtolower() deals with mixed case input');  
$t->is(strtolower(''), 'foo',  
    'strtolower() transforms empty strings into foo');
```

Listing 15-1

Launch the test set from the command line with the `test:unit` task. The command-line output is very explicit, and it helps you localize which tests failed and which passed. See the output of the example test in Listing 15-2.

Listing 15-2 - Launching a Single Unit Test from the Command Line

Listing 15-2

```
$ php symfony test:unit strtolower
1..7
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
ok 3 - strtolower() leaves lowercase characters unchanged
ok 4 - strtolower() leaves non alphabetical characters unchanged
ok 5 - strtolower() leaves blanks alone
ok 6 - strtolower() deals with mixed case input
not ok 7 - strtolower() transforms empty strings into foo
#     Failed test (.\\batch\\test.php at line 21)
#         got: ''
#     expected: 'foo'
# Looks like you failed 1 tests of 7.
```



The `include` statement at the beginning of Listing 15-1 is optional, but it makes the test file an independent PHP script that you can execute without the `symfony` command line, by calling `php test/unit/striplowerTest.php`.

Unit Testing Methods

The `lime_test` object comes with a large number of testing methods, as listed in Table 15-2.

Table 15-2 - Methods of the lime_test Object for Unit Testing

Method	Description
<code>diag(\$msg)</code>	Outputs a diag message but runs no test
<code>ok(\$test[, \$msg])</code>	Tests a condition and passes if it is true
<code>is(\$value1, \$value2[, \$msg])</code>	Compares two values and passes if they are equal (<code>==</code>)
<code>isnt(\$value1, \$value2[, \$msg])</code>	Compares two values and passes if they are not equal
<code>like(\$string, \$regexp[, \$msg])</code>	Tests a string against a regular expression
<code>unlike(\$string, \$regexp[, \$msg])</code>	Checks that a string doesn't match a regular expression
<code>cmp_ok(\$value1, \$operator, \$value2[, \$msg])</code>	Compares two arguments with an operator
<code>isa_ok(\$variable, \$type[, \$msg])</code>	Checks the type of an argument
<code>isa_ok(\$object, \$class[, \$msg])</code>	Checks the class of an object
<code>can_ok(\$object, \$method[, \$msg])</code>	Checks the availability of a method for an object or a class
<code>is_deeply(\$array1, \$array2[, \$msg])</code>	Checks that two arrays have the same values

Method	Description
<code>include_ok(\$file[, \$msg])</code>	Validates that a file exists and that it is properly included
<code>fail([\$msg])</code>	Always fails—useful for testing exceptions
<code>pass([\$msg])</code>	Always passes—useful for testing exceptions
<code>skip([\$msg, \$nb_tests])</code>	Counts as \$nb_tests tests—useful for conditional tests
<code>todo([\$msg])</code>	Counts as a test—useful for tests yet to be written
<code>comment(\$msg)</code>	Outputs a comment message but runs no test
<code>error(\$msg)</code>	Outputs a error message but runs no test
<code>info(\$msg)</code>	Outputs a info message but runs no test

The syntax is quite straightforward; notice that most methods take a message as their last parameter. This message is displayed in the output when the test passes. Actually, the best way to learn these methods is to test them, so have a look at Listing 15-3, which uses them all.

Listing 15-3 - Testing Methods of the lime_test Object, in test/unit/exampleTest.php

```
<?php
include dirname(__FILE__).'../../bootstrap/unit.php';

// Stub objects and functions for test purposes
class myObject
{
    public function myMethod()
    {
    }
}

function throw_an_exception()
{
    throw new Exception('exception thrown');
}

// Initialize the test object
$t = new lime_test(16);

$t->diag('hello world');
$t->ok(1 == '1', 'the equal operator ignores type');
$t->is(1, '1', 'a string is converted to a number for comparison');
$t->isnt(0, 1, 'zero and one are not equal');
$t->like('test01', '/test\d+/', 'test01 follows the test numbering pattern');
$t->unlike('tests01', '/test\d+/', 'tests01 does not follow the pattern');
$t->cmp_ok(1, '<', 2, 'one is inferior to two');
$t->cmp_ok(1, '!==', true, 'one and true are not identical');
$t->isa_ok('foobar', 'string', '\'foobar\' is a string');
$t->isa_ok(new myObject(), 'myObject', 'new creates object of the right class');
$t->can_ok(new myObject(), 'myMethod', 'objects of class myObject do have a myMethod method');
```

*Listing
15-3*

```
$array1 = array(1, 2, array(1 => 'foo', 'a' => '4'));
$t->is_deeply($array1, array(1, 2, array(1 => 'foo', 'a' => '4')),
    'the first and the second array are the same');
$t->include_ok('./fooBar.php', 'the fooBar.php file was properly
included');

try
{
    throw_an_exception();
    $t->fail('no code should be executed after throwing an exception');
}
catch (Exception $e)
{
    $t->pass('exception caught successfully');
}

if (!isset($foobar))
{
    $t->skip('skipping one test to keep the test count exact in the
condition', 1);
}
else
{
    $t->ok($foobar, 'foobar');
}

$t->todo('one test left to do');
```

You will find a lot of other examples of the usage of these methods in the symfony unit tests.



You may wonder why you would use `is()` as opposed to `ok()` here. The error message output by `is()` is much more explicit; it shows both members of the test, while `ok()` just says that the condition failed.

Testing Parameters

The initialization of the `lime_test` object takes as its first parameter the number of tests that should be executed. If the number of tests finally executed differs from this number, the lime output warns you about it. For instance, the test set of Listing 15-3 outputs as Listing 15-4. The initialization stipulated that 16 tests were to run, but only 15 actually took place, so the output indicates this.

Listing 15-4 - The Count of Test Run Helps You to Plan Tests

Listing 15-4 \$ php symfony test:unit example

```
1..16
# hello world
ok 1 - the equal operator ignores type
ok 2 - a string is converted to a number for comparison
ok 3 - zero and one are not equal
ok 4 - test01 follows the test numbering pattern
ok 5 - tests01 does not follow the pattern
ok 6 - one is inferior to two
ok 7 - one and true are not identical
ok 8 - 'foobar' is a string
ok 9 - new creates object of the right class
```

```

ok 10 - objects of class myObject do have a myMethod method
ok 11 - the first and the second array are the same
not ok 12 - the fooBar.php file was properly included
#     Failed test (.\\test\\unit\\testTest.php at line 27)
#     Tried to include './fooBar.php'
ok 13 - exception catched successfully
ok 14 # SKIP skipping one test to keep the test count exact in the
condition
ok 15 # TODO one test left to do
# Looks like you planned 16 tests but only ran 15.
# Looks like you failed 1 tests of 16.

```

The `diag()` method doesn't count as a test. Use it to show comments, so that your test output stays organized and legible. On the other hand, the `todo()` and `skip()` methods count as actual tests. A `pass()/fail()` combination inside a `try/catch` block counts as a single test.

A well-planned test strategy must contain an expected number of tests. You will find it very useful to validate your own test files—especially in complex cases where tests are run inside conditions or exceptions. And if the test fails at some point, you will see it quickly because the final number of run tests won't match the number given during initialization.

The `test:unit` Task

The `test:unit` task, which launches unit tests from the command line, expects either a list of test names or a file pattern. See Listing 15-5 for details.

Listing 15-5 - Launching Unit Tests

```

// Test directory structure
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php

$ php symfony test:unit myFunction          ## Run
myFunctionTest.php
$ php symfony test:unit myFunction mySecondFunction ## Run both tests
$ php symfony test:unit foo/*                ## Run barTest.php
$ php symfony test:unit *                   ## Run all tests
(recursive)

```

*Listing
15-5*

Stubs, Fixtures, and Autoloading

In a unit test, the autoloading feature is not active by default. Each class that you use in a test must be either defined in the test file or required as an external dependency. That's why many test files start with a group of `include` lines, as Listing 15-6 demonstrates.

Listing 15-6 - Including Classes in Unit Tests

```

<?php

include dirname(__FILE__).'../bootstrap/unit.php';
require_once sfConfig::get('sf_symfony_lib_dir').'/util/
sfToolkit.class.php';

$t = new lime_test(7);

```

*Listing
15-6*

```
// isPathAbsolute()
$t->diag('isPathAbsolute()');
$t->is(sfToolkit::isPathAbsolute('/test'), true,
      'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('\\test'), true,
      'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('C:\\test'), true,
      'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('d:/test'), true,
      'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('test'), false,
      'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../test'), false,
      'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('..\\test'), false,
      'isPathAbsolute() returns false if path is relative');
```

In unit tests, you need to instantiate not only the object you're testing, but also the object it depends upon. Since unit tests must remain unitary, depending on other classes may make more than one test fail if one class is broken. In addition, setting up real objects can be expensive, both in terms of lines of code and execution time. Keep in mind that speed is crucial in unit testing because developers quickly tire of a slow process.

Whenever you start including many scripts for a unit test, you may need a simple autoloading system. For this purpose, the `sfSimpleAutoload` class (which must be manually included) provides an `addDirectory()` method which expects an absolute path as parameter and that can be called several times in case you need to include several directories on the search path. All the classes located under this path will be autoloaded. For instance, if you want to have all the classes located under `sfConfig::get('sf_symfony_lib_dir')/util/` autoloaded, start your unit test script as follows:

Listing 15-7

```
require_once sfConfig::get('sf_symfony_lib_dir').'/autoload/
sfSimpleAutoload.class.php';
$autoload = sfSimpleAutoload::getInstance();
$autoload->addDirectory(sfConfig::get('sf_symfony_lib_dir').'/util');
$autoload->register();
```

Another good workaround for the autoloading issues is the use of stubs. A stub is an alternative implementation of a class where the real methods are replaced with simple canned data. It mimics the behavior of the real class, but without its cost. A good example of stubs is a database connection or a web service interface. In Listing 15-7, the unit tests for a mapping API rely on a `WebService` class. Instead of calling the real `fetch()` method of the actual web service class, the test uses a stub that returns test data.

Listing 15-7 - Using Stubs in Unit Tests

Listing 15-8

```
require_once dirname(__FILE__).'/../../lib/WebService.class.php';
require_once dirname(__FILE__).'/../../lib/MapAPI.class.php';

class testWebService extends WebService
{
    public static function fetch()
    {
        return file_get_contents(dirname(__FILE__).'/fixtures/data/
fake_web_service.xml');
    }
}
```

```
$myMap = new MapAPI();

$t = new lime_test(1, new lime_output_color());

$t->is($myMap->getMapSize(testWebService::fetch(), 100));
```

The test data can be more complex than a string or a call to a method. Complex test data is often referred to as fixtures. For coding clarity, it is often better to keep fixtures in separate files, especially if they are used by more than one unit test file. Also, don't forget that symfony can easily transform a YAML file into an array with the `sfYAML::load()` method. This means that instead of writing long PHP arrays, you can write your test data in a YAML file, as in Listing 15-8.

Listing 15-8 - Using Fixture Files in Unit Tests

```
// In fixtures.yml:
-
  input:  '/test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  '\\\\test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'C:\\\\test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'd:/test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'test'
  output: false
  comment: isPathAbsolute() returns false if path is relative
-
  input:  '../test'
  output: false
  comment: isPathAbsolute() returns false if path is relative
-
  input:  '..\\\\test'
  output: false
  comment: isPathAbsolute() returns false if path is relative

// In testTest.php
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
require_once sfConfig::get('sf_symfony_lib_dir').'/util/
sfToolkit.class.php';
require_once sfConfig::get('sf_symfony_lib_dir').'/yaml/sfYaml.class.php';

$testCases = sfYaml::load(dirname(__FILE__).'/fixtures.yml');

$t = new lime_test(count($testCases), new lime_output_color());
```

*Listing
15-9*

```
// isPathAbsolute()
$t->diag('isPathAbsolute()');
foreach ($testCases as $case)
{
    $t->is(sfToolkit::isPathAbsolute($case['input']),
$case['output'],$case['comment']);
}
```

Unit testing ORM classes

Testing Propel or Doctrine classes is a bit more involving as the generated objects rely on a long cascade of classes. Moreover, you need to provide a valid database connection to Propel and you also need to feed the database with some test data.

Thankfully, it is quite easy as symfony already provides everything you need:

- To get autoloading, you need to initialize a configuration object
- To get a database connection, you need to initialize the `sfDatabaseManager` class
- To load some test data, you can use the `sfPropelData` class

A typical Propel test file is shown in Listing 15-9.

Listing 15-9 - Testing Propel classes

Listing 15-10

```
<?php

include dirname(__FILE__).'./../bootstrap/unit.php';

new sfDatabaseManager($configuration);
$loader = new sfPropelData();
$loader->loadData(sfConfig::get('sf_data_dir').'/fixtures');

$t = new lime_test(1, new lime_output_color());

// begin testing your model class
$t->diag('->retrieveByUsername()');
$user = UserPeer::retrieveByUsername('fabien');
$t->is($user->getLastName(), 'Potencier', '->retrieveByUsername() returns
the User for the given username');
```

A typical Doctrine test file is shown in Listing 15-10.

Listing 15-10 - Testing Doctrine classes

Listing 15-11

```
<?php

include dirname(__FILE__).'./../bootstrap/unit.php';

new sfDatabaseManager($configuration);
Doctrine_Core::loadData(sfConfig::get('sf_data_dir').'/fixtures');

$t = new lime_test(1, new lime_output_color());

// begin testing your model class
$t->diag('->retrieveByUsername()');
$user = Doctrine::getTable('User')->findOneByUsername('fabien');
$t->is($user->getLastName(), 'Potencier', '->findOneByUsername() returns
the User for the given username');
```

Functional Tests

Functional tests validate parts of your applications. They simulate a browsing session, make requests, and check elements in the response, just like you would do manually to validate that an action does what it's supposed to do. In functional tests, you run a scenario corresponding to a use case.

What Do Functional Tests Look Like?

You could run your functional tests with a text browser and a lot of regular expression assertions, but that would be a great waste of time. Symfony provides a special object, called `sfBrowser`, which acts like a browser connected to a symfony application without actually needing a server—and without the slowdown of the HTTP transport. It gives access to the core objects of each request (the request, session, context, and response objects). Symfony also provides a `sfTestFunctional`, designed especially for functional tests, which takes a `sfBrowser` object and adds some smart assert methods.

A functional test traditionally starts with an initialization of a test browser object. This object makes a request to an action and verifies that some elements are present in the response.

For example, every time you generate a module skeleton with the `generate:module` or the `propel:generate-module` tasks, symfony creates a simple functional test for this module. The test makes a request to the default action of the module and checks the response status code, the module and action calculated by the routing system, and the presence of a certain sentence in the response content. For a `foobar` module, the generated `foobarActionsTest.php` file looks like Listing 15-11.

Listing 15-11 - Default Functional Test for a New Module, in tests/functional/frontend/foobarActionsTest.php

```
<?php
include dirname(__FILE__).'/../../bootstrap/functional.php';

$browser = new sfTestFunctional(new sfBrowser());

$browser->
    get('/foobar/index')->

    with('request')->begin()->
        isParameter('module', 'foobar')->
        isParameter('action', 'index')->
    end()->

    with('response')->begin()->
        isStatusCode(200)->
        checkElement('body', '/This is a temporary page/')->
    end()
;
```

Listing 15-12



The browser methods return an `sfTestFunctional` object, so you can chain the method calls for more readability of your test files. This is called a fluid interface to the object, because nothing stops the flow of method calls.

A functional test can contain several requests and more complex assertions; you will soon discover all the possibilities in the upcoming sections.

To launch a functional test, use the `test:functional` task with the `symfony` command line, as shown in Listing 15-12. This task expects an application name and a test name (omit the `Test.php` suffix).

Listing 15-12 - Launching a Single Functional Test from the Command Line

```
Listing 15-13 $ php symfony test:functional frontend foobarActions
# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
not ok 4 - response selector body does not match regex /This is a
temporary page/
# Looks like you failed 1 tests of 4.
1..4
```

The generated functional tests for a new module don't pass by default. This is because in a newly created module, the `index` action forwards to a congratulations page (included in the `symfony` default module), which contains the sentence "This is a temporary page". As long as you don't modify the `index` action, the tests for this module will fail, and this guarantees that you cannot pass all tests with an unfinished module.



In functional tests, the autoloading is activated, so you don't have to include the files by hand.

Browsing with the `sfBrowser` Object

The browser is capable of making GET and POST requests. In both cases, use a real URI as parameter. Listing 15-13 shows how to write calls to the `sfBrowser` object to simulate requests.

Listing 15-13 - Simulating Requests with the `sfBrowser` Object

```
Listing 15-14 include dirname(__FILE__).'/../../bootstrap/functional.php';

// Create a new browser
$b = new sfBrowser();

$b->get('/foobar/show/id/1'); // GET request
$b->post('/foobar/show', array('id' => 1)); // POST request

// The get() and post() methods are shortcuts to the call() method
$b->call('/foobar/show/id/1', 'get');
$b->call('/foobar/show', 'post', array('id' => 1));

// The call() method can simulate requests with any method
$b->call('/foobar/show/id/1', 'head');
$b->call('/foobar/add/id/1', 'put');
$b->call('/foobar/delete/id/1', 'delete');
```

A typical browsing session contains not only requests to specific actions, but also clicks on links and on browser buttons. As shown in Listing 15-14, the `sfBrowser` object is also capable of simulating those.

Listing 15-14 - Simulating Navigation with the `sfBrowser` Object

```
$b->get('/');
$b->get('/foobar/show/id/1');           // Request to the home page
$b->back();                            // Back to one page in history
$b->forward();                          // Forward one page in history
$b->reload();                           // Reload current page
$b->click('go');                        // Look for a 'go' link or button and click it
```

Listing
15-15

The browser handles a stack of calls, so the `back()` and `forward()` methods work as they do on a real browser.



The browser has its own mechanisms to manage sessions (`sfTestStorage`) and cookies.

Among the interactions that most need to be tested, those associated with forms probably rank first. To simulate form input and submission, you have three choices. You can either make a POST request with the parameters you wish to send, call `click()` with the form parameters as an array, or fill in the fields one by one and click the submit button. They all result in the same POST request anyhow. Listing 15-15 shows an example.

Listing 15-15 - Simulating Form Input with the sfBrowser Object

```
// Example template in modules/foobar/templates/editSuccess.php
<?php echo form_tag('foobar/update') ?>
  <input type="hidden" name="id" value="<?php echo $sf_params->get('id') ?>" />
  <input type="text" name="name" value="foo" />
  <input type="submit" value="go" />
  <textarea name="text1">foo</textarea>
  <textarea name="text2">bar</textarea>
</form>

// Example functional test for this form
$b = new sfBrowser();
$b->get('/foobar/edit/id/1');

// Option 1: POST request
$b->post('/foobar/update', array('id' => 1, 'name' => 'dummy', 'commit' => 'go'));

// Option 2: Click the submit button with parameters
$b->click('go', array('name' => 'dummy'));

// Option 3: Enter the form values field by field name then click the submit button
$b->setField('name', 'dummy')->
  click('go');
```

Listing
15-16

With the second and third options, the default form values are automatically included in the form submission, and the form target doesn't need to be specified.

When an action finishes by a `redirect()`, the browser doesn't automatically follow the redirection; you must follow it manually with `followRedirect()`, as demonstrated in Listing 15-16.

Listing 15-16 - The Browser Doesn't Automatically Follow Redirects

Listing 15-17

```
// Example action in modules/foobar/actions/actions.class.php
public function executeUpdate($request)
{
    // ...

    $this->redirect('foobar/show?id='.$request->getParameter('id'));

    // Example functional test for this action
    $b = new sfBrowser();
    $b->get('/foobar/edit?id=1')->
        click('go', array('name' => 'dummy'))->
        followRedirect();      // Manually follow the redirection
}
```

There is one last method you should know about that is useful for browsing: `restart()` reinitializes the browsing history, session, and cookies—as if you restarted your browser.

Once it has made a first request, the `sfBrowser` object can give access to the request, context, and response objects. It means that you can check a lot of things, ranging from the text content to the response headers, the request parameters, and configuration:

Listing 15-18

```
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();
```

Using Assertions

Due to the `sfBrowser` object having access to the response and other components of the request, you can do tests on these components. You could create a new `lime_test` object for that purpose, but fortunately `sfTestFunctional` proposes a `test()` method that returns a `lime_test` object where you can call the unit assertion methods described previously. Check Listing 15-17 to see how to do assertions via `sfTestFunctional`.

Listing 15-17 - The Test Browser Provides Testing Abilities with the test() Method

Listing 15-19

```
$b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit/id/1');
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();

// Get access to the lime_test methods via the test() method
$b->test()->is($request->getParameter('id'), 1);
$b->test()->is($response->getStatusCode(), 200);
$b->test()->is($response->getHttpHeader('content-type'), 'text/html; charset=utf-8');
$b->test()->like($response->getContent(), '/edit/');
```



The `getResponse()`, `getContext()`, `getRequest()`, and `test()` methods don't return an `sfBrowser` object, therefore you can't chain other `sfBrowser` method calls after them.

You can check incoming and outgoing cookies easily via the request and response objects, as shown in Listing 15-16.

Listing 15-16 - Testing Cookies with sfBrowser

Listing 15-20

```
$b->test()->is($request->getCookie('foo'), 'bar');      // Incoming cookie
$cookies = $response->getCookies();
$b->test()->is($cookies['foo'], 'foo=bar');           // Outgoing cookie
```

Using the `test()` method to test the request elements ends up in long lines. Fortunately, `sfTestFunctional` contains a bunch of proxy methods that help you keep your functional tests readable and short—in addition to returning an `sfTestFunctional` object themselves. For instance, you can rewrite Listing 15-15 in a faster way, as shown in Listing 15-18.

Listing 15-18 - Testing Directly with sfTestFunctional

```
$b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit/id/1')->
with('request')->isParameter('id', 1)->
with('response')->begin()->
  isStatusCode()->
  isHeader('content-type', 'text/html; charset=utf-8')->
  matches('/edit/')->
end()
;
```

*Listing
15-21*

Each proxy method is part of a tester group. Using a tester group is done by wrapping the method calls with the `with()` and `end()` methods. The `with()` methods takes the tester group name (like `request` and `response`).

The status 200 is the default value of the parameter expected by `isStatusCode()`, so you can call it without any argument to test a successful response.

One more advantage of proxy methods is that you don't need to specify an output text as you would with a `lime_test` method. The messages are generated automatically by the proxy methods, and the test output is clear and readable.

```
# get /foobar/edit/id/1
ok 1 - request parameter "id" is "1"
ok 2 - status code is "200"
ok 3 - response header "content-type" is "text/html"
ok 4 - response matches "/edit/"
1..4
```

*Listing
15-22*

In practice, the proxy methods of Listing 15-17 cover most of the usual tests, so you will seldom use the `test()` method on an `sfTestFunctional` object.

Listing 15-15 showed that `sfBrowser` doesn't automatically follow redirections. This has one advantage: You can test a redirection. For instance, Listing 15-19 shows how to test the response of Listing 15-14.

Listing 15-19 - Testing Redirections with sfTestFunctional

```
$b = new sfTestFunctional(new sfBrowser());
$b->
  get('/foobar/edit/id/1')->
  click('go', array('name' => 'dummy'))->
  with('request')->begin()->
    isParameter('module', 'foobar')->
    isParameter('action', 'update')->
  end()->
  with('response')->begin()->
    isStatusCode(200)->
    isRedirected()-> // Check that the response is a redirect
  end()>
```

*Listing
15-23*

```

followRedirect()-> // Manually follow the redirection

with('request')->begin()->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'show')->
end()->
with('response')->isStatusCode(200)
;

```

Using CSS Selectors

Many of the functional tests validate that a page is correct by checking for the presence of text in the content. With the help of regular expressions in the `matches()` method, you can check displayed text, a tag's attributes, or values. But as soon as you want to check something deeply buried in the response DOM, regular expressions are not ideal.

That's why the `sfTestFunctional` object supports a `getResponseDom()` method. It returns a libXML2 DOM object, much easier to parse and test than a flat text. Refer to Listing 15-20 for an example of using this method.

Listing 15-20 - The Test Browser Gives Access to the Response Content As a DOM Object

```

Listing 15-24 $b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit/id/1');
$dom = $b->getResponseDom();
$b->test()->is($dom->getElementsByName('input')->item(1)->getAttribute('type'), 'text'

```

But parsing an HTML document with the PHP DOM methods is still not fast and easy enough. If you are familiar with the CSS selectors, you know that they are an even more powerful way to retrieve elements from an HTML document. Symfony provides a tool class called `sfDomCssSelector` that expects a DOM document as construction parameter. It has a `getValues()` method that returns an array of strings according to a CSS selector, and a `getElements()` method that returns an array of DOM elements. See an example in Listing 15-20.

Listing 15-21 - The Test Browser Gives Access to the Response Content As an sfDomCssSelector Object

```

Listing 15-25 $b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit/id/1');
$c = new sfDomCssSelector($b->getResponseDom())
$b->test()->is($c->getValues('form input[type="hidden"][value="1"]'), array(''));
$b->test()->is($c->getValues('form textarea[name="text1"]'), array('foo'));
$b->test()->is($c->getValues('form input[type="submit"]'), array(''));

```

In its constant pursuit for brevity and clarity, symfony provides a shortcut for this: the `checkElement()` proxy method of the `response` tester group. This method makes Listing 15-21 look like Listing 15-22.

Listing 15-22 - The Test Browser Gives Access to the Elements of the Response by CSS Selectors

```

Listing 15-26 $b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit/id/1')-
    with('response')->begin()->
        checkElement('form input[type="hidden"][value="1"]', true)->
        checkElement('form textarea[name="text1"]', 'foo')-

```

```

    checkElement('form input[type="submit"]', 1) ->
end()
;

```

The behavior of the `checkElement()` method depends on the type of the second argument that it receives:

- If it is a Boolean, it checks that an element matching the CSS selector exists.
- If it is an integer, it checks that the CSS selector returns this number of results.
- If it is a regular expression, it checks that the first element found by the CSS selector matches it.
- If it is a regular expression preceded by `!`, it checks that the first element doesn't match the pattern.
- For other cases, it compares the first element found by the CSS selector with the second argument as a string.

The method accepts a third optional parameter, in the shape of an associative array. It allows you to have the test performed not on the first element returned by the selector (if it returns several), but on another element at a certain position, as shown in Listing 15-23.

Listing 15-23 - Using the Position Option to Match an Element at a Certain Position

```
$b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit?id=1')->
with('response')->begin()->
    checkElement('form textarea', 'foo')->
    checkElement('form textarea', 'bar', array('position' => 1))->
end()
;
```

*Listing
15-27*

The options array can also be used to perform two tests at the same time. You can test that there is an element matching a selector and how many there are, as demonstrated in Listing 15-24.

Listing 15-24 - Using the Count Option to Count the Number of Matches

```
$b = new sfTestFunctional(new sfBrowser());
$b->get('/foobar/edit?id=1')->
with('response')->checkElement('form input', true, array('count' => 3));
```

*Listing
15-28*

The selector tool is very powerful. It accepts most of the CSS 3 selectors, and you can use it for complex queries such as those of Listing 15-25.

Listing 15-25 - Example of Complex CSS Selectors Accepted by checkElement()

```
->checkElement('ul#list li a[href]', 'click me');
->checkElement('ul > li', 'click me');
->checkElement('ul + li', 'click me');
->checkElement('h1, h2', 'click me');
->checkElement('a[class$="foo"][href*="bar.html"]', 'my link');
->checkElement('p:last ul:nth-child(2) li:contains("Some text")');
```

*Listing
15-29*

Testing for errors

Sometimes, your actions or your model throw exceptions on purpose (for example to display a 404 page). Even if you can use a CSS selector to check for a specific error message in the generated HTML code, it's better to use the `throwsException` method to check that an exception has been thrown as shown in Listing 15-26.

Listing 15-26 - Testing for Exceptions

Listing 15-30

```
$b = new sfTestFunctional(new sfBrowser());
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    throwsException()-> // Checks that the last request
    threw an exception
    throwsException('RuntimeException')-> // Checks the class of the
exception
    throwsException(null, '/error/'); // Checks that the content of the
exception message matches the regular expression
```

Working in the Test Environment

The `sfTestFunctional` object uses a special front controller, set to the `test` environment. The default configuration for this environment appears in Listing 15-27.

Listing 15-27 - Default Test Environment Configuration, in frontend/config/settings.yml

Listing 15-31

```
test:
  .settings:
    error_reporting:      <?php echo ((E_ALL | E_STRICT) ^
E_NOTICE)."\n" ?>
    cache:                false
    web_debug:             false
    no_script_name:        false
    etag:                 false
```

The cache and the web debug toolbar are set to `false` in this environment. However, the code execution still leaves traces in a log file, distinct from the `dev` and `prod` log files, so that you can check it independently (`myproject/log/frontend_test.log`). In this environment, the exceptions don't stop the execution of the scripts—so that you can run an entire set of tests even if one fails. You can have specific database connection settings, for instance, to use another database with test data in it.

Before using the `sfBrowser` object, you have to initialize it. If you need to, you can specify a hostname for the application and an IP address for the client—that is, if your application makes controls over these two parameters. Listing 15-28 demonstrates how to do this.

Listing 15-28 - Setting Up the Browser with Hostname and IP

Listing 15-32

```
$b = new sfBrowser('myapp.example.com', '123.456.789.123');
```

The `test:functional` Task

The `test:functional` task can run one or more functional tests, depending on the number of arguments received. The rules look much like the ones of the `test:unit` task, except that the functional test task always expects an application as first argument, as shown in Listing 15-29.

Listing 15-29 - Functional Test Task Syntax

Listing 15-33

```
// Test directory structure
test/
  functional/
    frontend/
      myModuleActionsTest.php
```

```

myScenarioTest.php
backend/
myOtherScenarioTest.php

## Run all functional tests for one application, recursively
$ php symfony test:functional frontend

## Run one given functional test
$ php symfony test:functional frontend myScenario

## Run several tests based on a pattern
$ php symfony test:functional frontend my*

```

Test Naming Practices

This section lists a few good practices to keep your tests organized and easy to maintain. The tips concern file organization, unit tests, and functional tests.

As for the file structure, you should name the unit test files using the class they are supposed to test, and name the functional test files using the module or the scenario they are supposed to test. See Listing 15-30 for an example. 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 follow these guidelines.

Listing 15-30 - Example File Naming Practice

```

test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php
  functional/
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php

```

*Listing
15-34*

For unit tests, a good practice is to group the tests by function or method, and start each test group with a `diag()` call. The messages of each unit test should contain the name of the function or method tested, followed by a verb and a property, so that the test output looks like a sentence describing a property of the object. Listing 15-31 shows an example.

Listing 15-31 - Example Unit Test Naming Practice

```

// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string', 'strtolower() returns a string');
$t->is(strtolower('FOO'), 'foo', 'strtolower() transforms the input to
lowercase');

# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase

```

*Listing
15-35*

Functional tests should be grouped by page and start with a request. Listing 15-32 illustrates this practice.

Listing 15-32 - Example Functional Test Naming Practice

Listing 15-36

```
$browser->
    get('/foobar/index')->
    with('request')->begin()->
        isParameter('module', 'foobar')->
        isParameter('action', 'index')->
    end()->
    with('response')->begin()->
        isStatusCode(200)->
        checkElement('body', '/foobar/')->
    end()
;

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
ok 4 - response selector body matches regex /foobar/
```

If you follow this convention, the output of your test will be clean enough to use as a developer documentation of your project—enough so in some cases to make actual documentation useless.

Special Testing Needs

The unit and functional test tools provided by symfony should suffice in most cases. A few additional techniques are listed here to resolve common problems in automated testing: launching tests in an isolated environment, accessing a database within tests, testing the cache, and testing interactions on the client side.

Executing Tests in a Test Harness

The `test:unit` and `test:functional` tasks can launch a single test or a set of tests. But if you call these tasks without any parameter, they launch all the unit and functional tests written in the `test/` directory. A particular mechanism is involved to isolate each test file in an independent sandbox, to avoid contamination risks between tests. Furthermore, as it wouldn't make sense to keep the same output as with single test files in that case (the output would be thousands of lines long), the tests results are compacted into a synthetic view. That's why the execution of a large number of test files uses a test harness, that is, an automated test framework with special abilities. A test harness relies on a component of the lime framework called `lime_harness`. It shows a test status file by file, and an overview at the end of the number of tests passed over the total, as you see in Listing 15-33.

Listing 15-33 - Launching All Tests in a Test Harness

Listing 15-37

```
$ php symfony test:all

unit/myFunctionTest.php.....ok
unit/mySecondFunctionTest.php.....ok
unit/foo/barTest.php.....not ok

Failed Test                  Stat Total Fail List of Failed
-----
unit/foo/barTest.php          0     2     2   62 63
Failed 1/3 test scripts, 66.66% okay. 2/53 subtests failed, 96.22% okay.
```

The tests are executed the same way as when you call them one by one, only the output is made shorter to be really useful. In particular, the final chart focuses on the failed tests and helps you locate them.

You can launch all the tests with one call using the `test:all` task, which also uses a test harness, as shown in Listing 15-34. This is something that you should do before every transfer to production, to ensure that no regression has appeared since the latest release.

Listing 15-34 - Launching All the Tests of a Project

```
$ php symfony test:all
```

*Listing
15-38*

Accessing a Database

Unit tests often need to access a database. A database connection is automatically initialized when you call `sfBrowser::get()` for the first time. However, if you want to access the database even before using `sfBrowser`, you have to initialize a `sfDatabaseManager` object manually, as in Listing 15-35.

Listing 15-35 - Initializing a Database in a Test

```
$databaseManager = new sfDatabaseManager($configuration);
$databaseManager->loadConfiguration();

// Optionally, you can retrieve the current database connection
$con = Propel::getConnection();
```

*Listing
15-39*

You should populate the database with fixtures before starting the tests. This can be done via the `sfPropelData` object. This object can load data from a file, just like the `propel:dataload` task, or from an array, as shown in Listing 15-36.

Listing 15-36 - Populating a Database from a Test File

```
$data = new sfPropelData();

// Loading data from file
$data->loadData(sfConfig::get('sf_data_dir').'/fixtures/test_data.yml');

// Loading data from array
$fixtures = array(
    'Article' => array(
        'article_1' => array(
            'title'      => 'foo title',
            'body'       => 'bar body',
            'created_at' => time(),
        ),
        'article_2'   => array(
            'title'      => 'foo foo title',
            'body'       => 'bar bar body',
            'created_at' => time(),
        ),
    ),
);
$data->loadDataFromArray($fixtures);
```

*Listing
15-40*

Then, use the Propel objects as you would in a normal application, according to your testing needs. Remember to include their files in unit tests (you can use `sfSimpleAutoload` class to automate it, as explained in a tip in the “Stubs, Fixtures, and Autoloading” section previously in this chapter). Propel objects are autoloaded in functional tests.

Testing the Cache

When you enable caching for an application, the functional tests should verify that the cached actions do work as expected.

The first thing to do is enable cache for the test environment (in the `settings.yml` file). Then, if you want to test whether a page comes from the cache or whether it is generated, you should use the `isCached()` test method provided by the `view_cache` tester group. Listing 15-37 demonstrates this method.

Listing 15-37 - Testing the Cache with the `isCached()` Method

```
Listing 15-41 <?php

include dirname(__FILE__).'/../../bootstrap/functional.php';

// Create a new test browser
$b = new sfTestFunctional(new sfBrowser());

$b->get('/mymodule');
$b->with('view_cache')->isCached(true);           // Checks that the response
comes from the cache
$b->with('view_cache')->isCached(true, true); // Checks that the cached
response comes with layout
$b->with('view_cache')->isCached(false);        // Checks that the response
doesn't come from the cache
```



You don't need to clear the cache at the beginning of a functional test; the bootstrap script does it for you.

Testing Interactions on the Client

The main drawback of the techniques described previously 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 input that a user would do and execute scripts on the client side. Usually, these tests are reproduced by hand, but they are very time consuming and prone to error.

The solution is called Selenium⁵², which is a test framework written entirely in JavaScript. It executes a set of actions on a page just like a regular user would, using the current browser window. The advantage over the `sfBrowser` object is that Selenium is capable of executing JavaScript in a page, so you can test even Ajax interactions with it.

Selenium is not bundled with symfony by default. To install it, you need to create a new `selenium/` directory in your `web/` directory, and in it unpack the content of the Selenium archive⁵³. 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.



Be careful not to transfer the `selenium/` directory to your production server, since it would be accessible by anyone having access to your web document root via the browser.

Selenium tests are written in HTML and stored in the `web/selenium/tests/` directory. For instance, Listing 15-38 shows a functional test where the home page is loaded, the link click

52. <http://seleniumhq.org/>

53. <http://seleniumhq.org/download/>

me is clicked, and the text “Hello, World” is looked for in the response. Remember that in order to access the application in the test environment, you have to specify the `frontend_test.php` front controller.

Listing 15-38 - A Sample Selenium Test, in web/selenium/test/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>/frontend_test.php</td>
<td>&nbsp;</td></tr>
  <tr><td>clickAndWait</td> <td>link=click me</td>
<td>&nbsp;</td></tr>
  <tr><td>assertTextPresent</td> <td>Hello, World!</td>
<td>&nbsp;</td></tr>
</tbody>
</table>
</body>
</html>
```

*Listing
15-42*

A test case is represented by an HTML document containing a table with three columns: command, target, and value. Not all commands take a value, however. In this case, either leave the column blank or use ` ` to make the table look better. Refer to the Selenium website for a complete list of commands.

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. Listing 15-39 shows how.

Listing 15-39 - Adding a Test File to the Test Suite, in web/selenium/test/TestSuite.html

```
...
<tr><td><a href='./testIndex.html'>My First Test</a></td></tr>
...
```

*Listing
15-43*

To run the test, simply browse to

<http://myapp.example.com/selenium/index.html>

*Listing
15-44*

Select Main Test Suite, click the button to run all tests, and watch your browser as it reproduces the steps that you have told it to do.



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 extension⁵⁴, 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

54. <http://seleniumhq.org/projects/ide/>

add assert-type tests by right-clicking in the browser window and selecting the appropriate check under Append Selenium Command in the pop-up menu.

You can save the test to an 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.



Don't forget to reinitialize the test data before launching the Selenium test.

Summary

Automated tests include unit tests to validate methods or functions and functional tests to validate features. Symfony relies on the lime testing framework for unit tests and provides the `sfBrowser` and `sfTestFunctional` classes especially for functional tests. They provide many assertion methods, from basic to the most advanced, like CSS selectors. Use the `symfony` command line to launch tests, either one by one (with the `test:unit` and `test:functional` tasks) or in a test harness (with the `test:all` task). When dealing with data, automated tests use fixtures and stubs, and this is easily achieved within symfony unit tests.

If you make sure to write enough unit tests to cover a large part of your applications (maybe using the TDD methodology), you will feel safer when refactoring internals or adding new features, and you may even gain some time on the documentation task.

Chapter 16

Application Management Tools

During both the development and deployment phases, developers require a consistent stream of diagnostic information in order to determine whether the application is working as intended. This information is generally aggregated through logging and debugging utilities. Because of the central role frameworks, such as symfony, play in driving applications, it's crucial that such capabilities are tightly integrated to ensure efficient developmental and operational activities.

During the life of an application on the production server, the application administrator repeats a large number of tasks, from log rotation to upgrades. A framework must also provide tools to automate these tasks as much as possible.

This chapter explains how symfony application management tools can answer all these needs.

Logging

The only way to understand what went wrong during the execution of a request is to review a trace of the execution process. Fortunately, as you'll learn in this section, both PHP and symfony tend to log large amounts of this sort of data.

PHP Logs

PHP has an `error_reporting` parameter, defined in `php.ini`, that specifies which PHP events are logged. Symfony allows you to override this value, per application and environment, in the `settings.yml` file, as shown in Listing 16-1.

Listing 16-1 - Setting the Error Reporting Level, in `frontend/config/settings.yml`

```
prod:
  .settings:
    error_reporting:  <?php echo (E_PARSE | E_COMPILE_ERROR | E_ERROR |
E_CORE_ERROR | E_USER_ERROR)."\\n" ?>

dev:
  .settings:
    error_reporting:  <?php echo (E_ALL | E_STRICT)."\\n" ?>
```

*Listing
16-1*

In order to avoid performance issues in the production environment, the server logs only the critical PHP errors. However, in the development environment, all types of events are logged, so that the developer can have all the information necessary to trace errors.

The location of the PHP log files depends on your `php.ini` configuration. If you never bothered about defining this location, PHP probably uses the logging facilities provided by

your web server (such as the Apache error logs). In this case, you will find the PHP logs under the web server log directory.

Symfony Logs

In addition to the standard PHP logs, symfony can log a lot of custom events. You can find all the symfony logs under the `myproject/log/` directory. There is one file per application and per environment. For instance, the development environment log file of the `frontend` application is named `frontend_dev.log`, the production one is named `frontend_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. Each line includes the exact time of the event, the nature of the event, the object being processed, and any additional relevant details. Listing 16-2 shows an example of symfony log file content.

Listing 16-2 - Sample Symfony Log File Content, in log/frontend_dev.log

```
Listing 16-2 Nov 15 16:30:25 symfony [info ] {sfAction} call
"barActions->executemessages()"
Nov 15 16:30:25 symfony [info ] {sfPropelLogger} executeQuery: SELECT
bd_message.ID...
Nov 15 16:30:25 symfony [info ] {sfView} set slot "leftbar" (bar/index)
Nov 15 16:30:25 symfony [info ] {sfView} set slot "messageblock" (bar/
mes...
Nov 15 16:30:25 symfony [info ] {sfView} execute view for template
"mess...
Nov 15 16:30:25 symfony [info ] {sfView} render "/home/production/
myproject/...
Nov 15 16:30:25 symfony [info ] {sfView} render to client
```

You can find many details in these files, including the actual SQL queries sent to the database, the templates called, the chain of calls between objects, and so on.



The format of the file logs is configurable by overriding the `format` and/or the `time_format` settings in `factories.yml` as shown in Listing 16-3.

Listing 16-3 - Changing the Log Format

```
Listing 16-3 all:
logger:
param:
sf_file_debug:
param:
format:      %time% %type% [%priority%] %message%%EOL%
time_format: %b %d %H:%M:%S
```

Symfony Log Level Configuration

There are eight levels of symfony log messages: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`, and `debug`, which are the same as the `PEAR::Log`⁵⁵ package levels. You can configure the maximum level to be logged in each environment in the `factories.yml` configuration file of each application, as demonstrated in Listing 16-4.

Listing 16-4 - Default Logging Configuration, in frontend/config/factories.yml

55. <http://pear.php.net/package/Log/>

```
prod:
  logger:
    param:
      level: err
```

*Listing
16-4*

By default, in all environments except the production environment, all the messages are logged (up to the least important level, the debug level). In the production environment, logging is disabled by default; if you change `logging_enabled` to `true` in `settings.yml`, only the most important messages (from `crit` to `emerg`) appear in the logs.

You can change the logging level in the `factories.yml` file for each environment to limit the type of logged messages.



To see if logging is enabled, call `sfConfig::get('sf_logging_enabled')`.

Adding a Log Message

You can manually add a message in the symfony log file from your code by using one of the techniques described in Listing 16-5.

Listing 16-5 - Adding a Custom Log Message

```
// From an action
$this->logMessage($message, $level);

// From a template
<?php use_helper('Debug') ?>

<?php log_message($message, $level) ?>
```

*Listing
16-5*

`$level` can have the same values as in the log messages.

Alternatively, to write a message in the log from anywhere in your application, use the `sfLogger` methods directly, as shown in Listing 16-6. The available methods bear the same names as the log levels.

Listing 16-6 - Adding a Custom Log Message from Anywhere

```
if (sfConfig::get('sf_logging_enabled'))
{
  sfContext::getInstance()->getLogger()->info($message);
}
```

*Listing
16-6*

Customizing the Logging

Symfony's logging system is very simple, yet it is also easy to customize. The only prerequisite is that logger classes must extend the `sfLogger` class, which defines a `doLog()` method. Symfony calls the `doLog()` method with two parameters: `$message` (the message to be logged), and `$priority` (the log level).

The `myLogger` class defines a simple logger using the PHP `error_log` function:

```
Listing 16-7 class myLogger extends sfLogger
{
    protected function doLog($message, $priority)
    {
        error_log(sprintf('%s (%s)', $message,
sfLogger::getPriorityName($priority)));
    }
}
```

To create a logger from an existing class, you can just implement the `sfLoggerInterface` interface, which defines a `log()` method. The method takes the same two parameters as the `doLog()` method:

```
Listing 16-8 require_once('Log.php');
require_once('Log/error_log.php');

// Define a thin wrapper to implement the interface
// for the logger we want to use with symfony
class Log_my_error_log extends Log_error_log implements sfLoggerInterface
{
}
```

Purging and Rotating Log Files

Don't forget to periodically purge the `log/` directory of your applications, because these files have the strange habit of growing by several megabytes in a few days, depending, of course, on your traffic. Symfony provides a special `log:clear` task for this purpose, which you can launch regularly by hand or put in a cron table. For example, the following command erases the symfony log files:

```
Listing 16-9 $ php symfony log:clear
```

For both better performance and security, you probably want to store symfony logs in several small files instead of one single large file. The ideal storage strategy for log files is to back up and empty the main log file regularly, but to keep only a limited number of backups. You can enable such a log rotation with a `period` of 7 days and a `history` (number of backups) of 10, as shown in Listing 16-7. You would work with one active log file plus ten backup files containing seven days' worth of history each. Whenever the next period of seven days ends, the current active log file goes into backup, and the oldest backup is erased.

Listing 16-7 - Launching Log Rotation

```
Listing 16-10 $ php symfony log:rotate frontend prod --period=7 --history=10
```

The backup log files are stored in the `logs/history/` directory and suffixed with the date they were saved.

Debugging

No matter how proficient a coder you are, you will eventually make mistakes, even if you use symfony. Detecting and understanding errors is one of the keys of fast application development. Fortunately, symfony provides several debug tools for the developer.

Symfony Debug Mode

Symfony has a debug mode that facilitates application development and debugging. When it is on, the following happens:

- The configuration is checked at each request, so a change in any of the configuration files has an immediate effect, without any need to clear the configuration cache.
- The error messages display the full stack trace in a clear and useful way, so that you can more efficiently find the faulty element.
- More debug tools are available (such as the detail of database queries).
- The Propel/Doctrine debug mode is also activated, so any error in a call to a Propel/Doctrine object will display a detailed chain of calls through the Propel/Doctrine architecture.

On the other hand, when the debug mode is off, processing is handled as follows:

- The YAML configuration files are parsed only once, then transformed into PHP files stored in the `cache/config/` folder. Every request after the first one ignores the YAML files and uses the cached configuration instead. As a consequence, the processing of requests is much faster.
- To allow a reprocessing of the configuration, you must manually clear the configuration cache.
- An error during the processing of the request returns a response with code 500 (Internal Server Error), without any explanation of the internal cause of the problem.

The debug mode is activated per application in the front controller. It is controlled by the value of the third argument passed to the `getApplicationConfiguration()` method call, as shown in Listing 16-8.

Listing 16-8 - Sample Front Controller with Debug Mode On, in web/frontend_dev.php

```
<?php  
  
require_once(dirname(__FILE__).'../config/  
ProjectConfiguration.class.php');  
  
$configuration =  
ProjectConfiguration::getApplicationConfiguration('frontend', 'dev', true);  
sfContext::createInstance($configuration)->dispatch();
```

*Listing
16-11*



In your production server, you should not activate the debug mode nor leave any front controller with debug mode on available. Not only will the debug mode slow down the page delivery, but it may also reveal the internals of your application. Even though the debug tools never reveal database connection information, the stack trace of exceptions is full of dangerous information for any ill-intentioned visitor.

Symfony Exceptions

When an exception occurs in the debug mode, symfony displays a useful exception notice that contains everything you need to find the cause of the problem.

The exception messages are clearly written and refer to the most probable cause of the problem. They often provide possible solutions to fix the problem, and for most common problems, the exception pages even contain a link to a symfony website page with more details about the exception. The exception page shows where the error occurred in the PHP code (with syntax highlighting), together with the full stack of method calls, as shown in Figure 16-1. You can follow the trace to the first call that caused the problem. The arguments that were passed to the methods are also shown.



Symfony really relies on PHP exceptions for error reporting. For instance, the 404 error can be triggered by an `sfError404Exception`.

Figure 16-1 - Sample exception message for a symfony application

The screenshot shows a detailed exception page from a Symfony application. At the top, it says "[sfException]" and has a warning icon. The main message is "The date is not in the expected UNIX timestamp format". Below this is a "stack trace" section with the following content:

```

[sfException]
The date is not in the expected UNIX timestamp format

stack trace
1. at ()
in SF_ROOT_DIR\apps\frontend\modules\test\templates\indexSuccess.php line 2 ...
1.
<?php use_helper('Date', 'Number', 'I18N') ?>
2. <?php throw new sfException("The date is not in the expected UNIX timestamp format" ...
3. <?php $now = time() ?>
4. <?php echo format_datetime($now) ?><br/>
5. <?php $sf_user->setCulture('en_US') ?>

2. at require()
in SF_ROOT_DIR\lib\symfony\view\sfPHPView.class.php line 109 ...
106. // render to variable
107. ob_start();
108. ob_implicit_flush(0);
109. require($sf_file);
110. $retval = ob_get_clean();
111.
112. return $retval;

3. at
sfPHPView->renderFile('C:\wamp\www\sf_sandbox\apps\frontend\modules/test/templates/indexSuccess.php' ...
in SF_ROOT_DIR\lib\symfony\view\sfPHPView.class.php line 240 ...
4. at sfPHPView->render()
in SF_ROOT_DIR\lib\symfony\filter\sfExecutionFilter.class.php line 170 ...
5. at sfExecutionFilter->execute(object('sfFilterChain'))
in SF_ROOT_DIR\lib\symfony\filter\sfFilterChain.class.php line 76 ...
6. at sfFilterChain->execute()
in SF_ROOT_DIR\lib\symfony\filter\sfFlashFilter.class.php line 50 ...

```

During the development phase, the symfony exceptions will be of great use as you debug your application.

Xdebug Extension

The Xdebug⁵⁶ PHP 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 this extension when you debug the application. The extension installation depends very much on your platform; refer to the Xdebug website for detailed

56. <http://xdebug.org/>

installation guidelines. Once Xdebug is installed, you need to activate it manually in your `php.ini` file after installation. For *nix platforms, this is done by adding the following line:

```
zend_extension="/usr/local/lib/php/extensions/no-debug-non-zts-20041030/
xdebug.so"
```

Listing 16-12

For Windows platforms, the Xdebug activation is triggered by this line:

```
extension=php_xdebug.dll
```

Listing 16-13

Listing 16-9 gives an example of Xdebug configuration, which must also be added to the `php.ini` file.

Listing 16-9 - Sample Xdebug Configuration

```
;xdebug.profiler_enable=1
;xdebug.profiler_output_dir="/tmp/xdebug"
xdebug.auto_trace=1           ; enable tracing
xdebug.trace_format=0
;xdebug.show_mem_delta=0      ; memory difference
;xdebug.show_local_vars=1
;xdebug.max_nesting_level=100
```

Listing 16-14

You must restart your web server for the Xdebug mode to be activated.



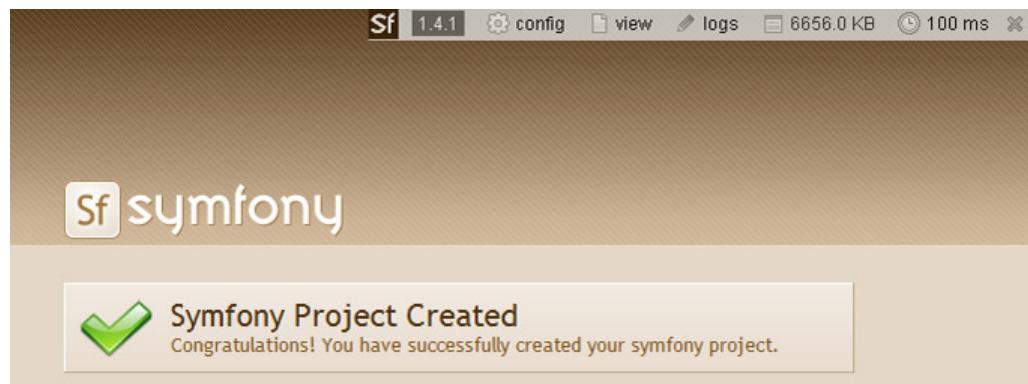
Don't forget to deactivate Xdebug mode in your production platform. Not doing so will slow down the execution of every page a lot.

Web Debug Toolbar

The log files contain interesting information, but they are not very easy to read. The most basic task, which is to find the lines logged for a particular request, can be quite tricky if you have several users simultaneously using an application and a long history of events. That's when you start to need a web debug toolbar.

This toolbar appears as a semitransparent box superimposed over the normal content in the browser, in the top-right corner of the window, as shown in Figure 16-2. It gives access to the symfony log events, the current configuration, the properties of the request and response objects, the details of the database queries issued by the request, and a chart of processing times related to the request.

Figure 16-2 - The web debug toolbar appears in the top-right corner of the window



The color of the debug toolbar background depends on the highest level of log message issued during the request. If no message passes the `debug` level, the toolbar has a gray background. If a single message reaches the `err` level, the toolbar has a red background.



Don't confuse the debug mode with the web debug toolbar. The debug toolbar can be displayed even when the debug mode is off, although, in that case, it displays much less information.

To activate the web debug toolbar for an application, open the `settings.yml` file and look for the `web_debug` key. In the `prod` and `test` environments, the default value for `web_debug` is `false`, so you need to activate it manually if you want it. In the `dev` environment, the default configuration has it set to `true`, as shown in Listing 16-10.

Listing 16-10 - Web Debug Toolbar Activation, in `frontend/config/settings.yml`

Listing 16-15

```
dev:
  .settings:
    web_debug: true
```

When displayed, the web debug toolbar offers a lot of information/interaction:

- Click the symfony logo to toggle the visibility of the toolbar. When reduced, the toolbar doesn't hide the elements located at the top of the page.
- Click the "config" section to show the details of the request, response, settings, globals, and PHP properties, as shown in Figure 16-3. The top line sums up the important configuration settings, such as the debug mode, the cache, and the presence of a PHP accelerator (they appear in red if they are deactivated and in green if they are activated).

Figure 16-3 - The "config" section shows all the variables and constants of the request

- When the cache is enabled, a green arrow appears in the toolbar. Click this arrow to reprocess the page, regardless of what is stored in the cache (but the cache is not cleared).
- Click the "logs" section to reveal the log messages for the current request, as shown in Figure 16-4. According to the importance of the events, they are displayed in gray, yellow, or red lines. You can filter the events that are displayed by category using the links displayed at the top of the list.

Figure 16-4 - The "logs" section shows the log messages for the current request

Log and debug messages		
[all]	[none]	Sf 1.0.0 vars & config logs & msgs 1 31 ms X
#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/:module/:action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list',)
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****',), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.66 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article/list"
16	View	render "sf_app_dir/modules/article/templates/listSuccess.php"
17	View	decorate content with "sf_app_dir/templates/layout.php"
18	View	render "sf_app_dir/templates/layout.php"

 When the current action results from a redirect, only the logs of the latest request are present in the “logs” pane, so the log files are still indispensable for good debugging.

- For requests executing SQL queries, a database icon appears in the toolbar. Click it to see the detail of the queries, as shown in Figure 16-5.
- To the right of a clock icon is the total time necessary to process the request. Be aware that the web debug toolbar and the debug mode slow down the request execution, so try to refrain from considering the timings per se, and pay attention to only the differences between the execution time of two pages. Click the clock icon to see details of the processing time category by category, as shown in Figure 16-6. Symfony displays the time spent on specific parts of the request processing. Only the times related to the current request make sense for an optimization, so the time spent in the symfony core is not displayed. That’s why these times don’t sum up to the total time.
- Click the red x at the right end of the toolbar to hide the toolbar.

Figure 16-5 - The database queries section shows queries executed for the current request

SQL queries

```

1. [1.00 ms] SELECT COUNT(ask_question.ID) FROM ask_question, ask_interest WHERE
ask_interest.CREATED_AT>'2006-11-24 10:47:17' AND ask_interest.QUESTION_ID=ask_question.ID
2. [3.53 ms] SELECT ask_question.ID, ask_question.USER_ID, ask_question.TITLE, ask_question.STRIPPED_TITLE,
ask_question.BODY, ask_question.HTML_BODY, ask_question.INTERESTED_USERS, ask_question.REPORTS,
ask_question.CREATED_AT, ask_question.UPDATED_AT, ask_user.ID, ask_user.NICKNAME,
ask_user.FIRST_NAME, ask_user.LAST_NAME, ask_user.EMAIL, ask_user.SHA1_PASSWORD, ask_user.SALT,
ask_user.HAS_PAYPAL, ask_user.WANT_TO_BE_MODERATOR, ask_user.IS_MODERATOR,
ask_user.IS_ADMINISTRATOR, ask_user.DELETIONS, ask_user.CREATED_AT, count(ask_interest.USER_ID) AS
count FROM ask_question, ask_user, ask_interest WHERE ask_interest.CREATED_AT>'2006-11-24 10:47:17' AND
ask_interest.QUESTION_ID=ask_question.ID AND ask_question.USER_ID=ask_user.ID GROUP BY
ask_interest.QUESTION_ID ORDER BY count DESC, ask_question.CREATED_AT DESC LIMIT 10
3. [0.49 ms] SELECT ask_answer.ID, ask_answer.QUESTION_ID, ask_answer.USER_ID, ask_answer.BODY,
ask_answer.HTML_BODY, ask_answer.RELEVANCY_UP, ask_answer.RELEVANCY_DOWN,
ask_answer.REPORTS, ask_answer.CREATED_AT FROM ask_answer WHERE ask_answer.QUESTION_ID=1
4. [0.45 ms] SELECT DISTINCT ask_question_tag.QUESTION_ID, ask_question_tag.USER_ID,
ask_question_tag.CREATED_AT, ask_question_tag.TAG, ask_question_tag.NORMALIZED_TAG,
UPPER(ask_question_tag.NORMALIZED_TAG) FROM ask_question_tag WHERE
ask_question_tag.QUESTION_ID=1 GROUP BY ask_question_tag.NORMALIZED_TAG ORDER BY
UPPER(ask_question_tag.NORMALIZED_TAG) ASC
5. [0.33 ms] SELECT ask_question_tag.NORMALIZED_TAG AS tag, COUNT(ask_question_tag.NORMALIZED_TAG) AS
count FROM ask_question_tag WHERE ask_question_tag.QUESTION_ID = 1 GROUP BY
ask_question_tag.NORMALIZED_TAG ORDER BY count DESC LIMIT 5
6. [0.43 ms] SELECT ask_answer.ID, ask_answer.QUESTION_ID, ask_answer.USER_ID, ask_answer.BODY,
ask_answer.HTML_BODY, ask_answer.RELEVANCY_UP, ask_answer.RELEVANCY_DOWN,
ask_answer.REPORTS, ask_answer.CREATED_AT FROM ask_answer WHERE ask_answer.QUESTION_ID=2

```

Figure 16-6 - The clock icon shows execution time by category

Timers

type	calls	time (ms)
Configuration	14	60.42
Action "question/frontpage"	1	132.43
Database	9	7.56
View "Success" for "question/frontpage"	1	243.24
Partial "question/_question_list"	1	151.49
Partial "question/_question_block"	2	119.74
Partial "question/_interested_user"	2	12.99
Partial "moderator/_question_options"	2	2.80
Component "sidebar/default"	1	0.02
Partial "sidebar/_default"	1	25.62
Partial "tag/_tag_cloud"	1	2.09
Partial "question/_search"	1	0.97
Partial "sidebar/_rss_links"	1	3.08
Partial "sidebar/_moderation"	1	1.32
Partial "sidebar/_administration"	1	1.85

Adding your own timer

Symfony uses the `sfTimer` class to calculate the time spent on the configuration, the model, the action, and the view. Using the same object, you can time a custom process and display the result with the other timers in the web debug toolbar. This can be very useful when you work on performance optimizations.

To initialize timing on a specific fragment of code, call the `getTimer()` method. It will return an `sfTimer` object and start the timing. Call the `addTime()` method on this object to stop the timing. The elapsed time is available through the `getElapsedTime()` method, and displayed in the web debug toolbar with the others.

```
// Initialize the timer and start timing
$timer = sfTimerManager::getTimer('myTimer');

// Do things
...

// Stop the timer and add the elapsed time
$timer->addTime();

// Get the result (and stop the timer if not already stopped)
$elapsedTime = $timer->getElapsedTime();
```

*Listing
16-16*

The benefit of giving a name to each timer is that you can call it several times to accumulate timings. For instance, if the `myTimer` timer is used in a utility method that is called twice per request, the second call to the `getTimer('myTimer')` method will restart the timing from the point calculated when `addTime()` was last called, so the timing will add up to the previous one. Calling `getCalls()` on the timer object will give you the number of times the timer was launched, and this data is also displayed in the web debug toolbar.

```
// Get the number of calls to the timer
$nbCalls = $timer->getCalls();
```

*Listing
16-17*

In Xdebug mode, the log messages are much richer. All the PHP script files and the functions that are called are logged, and symfony knows how to link this information with its internal log. Each line of the log messages table has a double-arrow button, which you can click to see further details about the related request. If something goes wrong, the Xdebug mode gives you the maximum amount of detail to find out why.



The web debug toolbar is not included by default in Ajax responses and documents that have a non-HTML content-type. For the other pages, you can disable the web debug toolbar manually from within an action by simply calling `sfConfig::set('sf_web_debug', false)`.

Manual Debugging

Getting access to the framework debug messages is nice, but being able to log your own messages is better. Symfony provides shortcuts, accessible from both actions and templates, to help you trace events and/or values during request execution.

Your custom log messages appear in the symfony log file as well as in the web debug toolbar, just like regular events. (Listing 16-5 gave an example of the custom log message syntax.) A custom message is a good way to check the value of a variable from a template, for instance. Listing 16-11 shows how to use the web debug toolbar for developer's feedback from a template (you can also use `$this->logMessage()` from an action).

Listing 16-11 - Inserting a Message in the Log for Debugging Purposes

Listing 16-18

```
<?php use_helper('Debug') ?>
...
<?php if ($problem): ?>
    <?php log_message('{sfAction} been there', 'err') ?>
    ...
<?php endif ?>
```

The use of the `err` level guarantees that the event will be clearly visible in the list of messages, as shown in Figure 16-7.

Figure 16-7 - A custom log message appears in the “logs” section of the web debug toolbar



The screenshot shows the "Logs" section of the Symfony web debug toolbar. The title bar says "Log and debug messages". Below it is a table with columns: #, type, and message. The table contains 19 rows of log entries. Row 17 is highlighted with a gray background, indicating it is the custom message from Listing 16-11. The message in row 17 is "been there".

#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/:module/:action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list',)
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****'), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.35 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article:list"
16	View	render "sf_app_dir/modules/article/templates/listSuccess.php"
17	Action	been there
18	View	decorate content with "sf_app_dir/templates/layout.php"
19	View	render "sf_app_dir/templates/layout.php"

Using symfony outside of a web context

You may want to execute a script from the command line (or via a cron table) with access to all the symfony classes and features, for instance to launch batch e-mail jobs or to periodically update your model through a process-intensive calculation. The simple way of doing this is to create a PHP script that reproduces the first steps of a front controller, so that symfony is properly initialized. You can also use the symfony command line system, to take advantage of arguments parsing and automated database initialization.

Batch Files

Initializing symfony just takes a couple lines of PHP code. You can take advantage of all symfony features by creating a PHP file, for instance under the `lib/` directory of your project, starting with the lines shown in Listing 16-12.

Listing 16-12 - Sample Batch Script, in lib/myScript.php

```
<?php
require_once(dirname(__FILE__).'/../config/
ProjectConfiguration.class.php');
$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'dev', true);

// Remove the following lines if you don't use the database layer
$databaseManager = new sfDatabaseManager($configuration);

// add code here
```

*Listing
16-19*

This strongly looks like the first lines of a front controller (see Chapter 6), because these lines do the same: initialize symfony, parse a project and an application configuration. Note that the `ProjectConfiguration::getApplicationConfiguration` method expects three parameters:

- an application name
- an environment name
- a Boolean, defining if the debug features should be enabled or not

To execute your code, just call the script from the command line:

```
$ php lib/myScript.php
```

*Listing
16-20*

Custom Tasks

An alternative way of creating custom command line scripts using symfony is to write a **symfony task**. Just like the `cache:clear` and the `propel:build-model` tasks, you can launch your own custom tasks from the command line with `php symfony`. Custom tasks benefit from the ability to parse command line arguments and options, can embed their own help text, and can extend existing tasks.

A custom task is just a class extending `sfBaseTask` and located under a `lib/task/` directory, either under the project root, or in a plugin directory. Its file name must end with '`Task.class.php`'. Listing 16-13 shows a sample custom task.

Listing 16-13 - Sample Task, in lib/task/testHelloTask.class.php

```
<?php
class testHelloTask extends sfBaseTask
{
    protected function configure()
    {
        $this->namespace = 'test';
        $this->name = 'hello';
        $this->briefDescription = 'Says hello';
    }

    protected function execute($arguments = array(), $options = array())
    {
        // your code here
        $this->log('Hello, world!');
    }
}
```

*Listing
16-21*

The code written in the `execute` method has access to all the symfony libraries, just like in the previous batch script. The difference is how you call the custom task:

Listing 16-22 \$ php symfony test:hello

The task name comes from the protected `namespace` and `name` properties (not from the class name, nor from the file name). And since your task is integrated into the `symfony` command line, it appears in the task list when you just type:

Listing 16-23 \$ php symfony

Rather than writing a task skeleton by yourself, you can use the `symfony generate:task` task. It creates an empty task, and has plenty of customization options. Make sure you check them by calling:

Listing 16-24 \$ php symfony help generate:task

Tasks can accept arguments (compulsory parameters, in a predefined order) and options (optional and unordered parameters). Listing 16-14 shows a more complete task, taking advantage of all these features.

Listing 16-14 - More Complete Sample Task, in lib/task/mySecondTask.class.php

```
Listing 16-25 class mySecondTask extends sfBaseTask
{
    protected function configure()
    {
        $this->namespace      = 'foo';
        $this->name            = 'mySecondTask';
        $this->briefDescription = 'Does some neat things, with style';
        $this->detailedDescription = <<<EOF
The [foo:mySecondTask|INFO] task manages the process of achieving things
for you.
Call it with:
```

[php symfony foo:mySecondTask frontend|INFO]

You can enable verbose output by using the [verbose|COMMENT] option:

```
[php symfony foo:mySecondTask frontend --verbose=on|INFO]
EOF;
    $this->addArgument('application', sfCommandArgument::REQUIRED, 'The
application name');
    $this->addOption('verbose', null, sfCommandOption::PARAMETER_REQUIRED,
'Enables verbose output', false);
}

protected function execute($arguments = array(), $options = array())
{
    // add code here
}
```



If your task needs access to the database layer through the `, it should extend sfPropelBaseTask instead of sfBaseTask. The task initialization will then take care of loading the additional Propel classes. You can start a database connection in the execute() method by calling:`

```
$databaseManager = new sfDatabaseManager($this->configuration);
```

If the task configuration defines an `application` and an `env` argument, they are automatically considered when building the task configuration, so that a task can use any of the database connections defined in your `databases.yml`. By default, skeletons generated by a call to `generate:task` include this initialization.

For more examples on the abilities of the task system, check the source of existing symfony tasks.

Populating a Database

In the process of application development, developers are often faced with the problem of database population. A few specific solutions exist for some database systems, but none can be used on top of the object-relational mapping. Thanks to YAML and the `sfPropelData` object, symfony can automatically transfer data from a text source to a database. Although writing a text file source for data may seem like more work than entering the records by hand using a CRUD interface, it will save you time in the long run. You will find this feature very useful for automatically storing and populating the test data for your application.

Fixture File Syntax

Symfony can read data files that follow a very simple YAML syntax, provided that they are located under the `data/fixtures/` directory. Fixture files are organized by class, each class section being introduced by the class name as a header. For each class, records labeled with a unique string are defined by a set of `fieldname: value` pairs. Listing 16-15 shows an example of a data file for database population.

Listing 16-15 - Sample Fixture File, in `data/fixtures/import_data.yml`

```
Article:                                ## Insert records in the blog_article
table
first_post:                            ## First record label
  title:      My first memories
  content: |
    For a long time I used to go to bed early. Sometimes, when I had put
    out my candle, my eyes would close so quickly that I had not even
time
  to say "I'm going to sleep".

second_post:                           ## Second record label
  title:      Things got worse
  content: |
    Sometimes he hoped that she would die, painlessly, in some accident,
    she who was out of doors in the streets, crossing busy thoroughfares,
    from morning to night.
```

Listing
16-26

Symfony translates the column keys into setter methods by using a camelCase converter (`setTitle()`, `setContent()`). This means that you can define a `password` key even if the actual table doesn't have a `password` field—just define a `setPassword()` method in the `User` object, and you can populate other columns based on the password (for instance, a hashed version of the password).

The primary key column doesn't need to be defined. Since it is an auto-increment field, the database layer knows how to determine it.

The `created_at` columns don't need to be set either, because symfony knows that fields named that way must be set to the current system time when created.

Launching the Import

The `propel:data-load` task imports data from YAML files to a database. The connection settings come from the `databases.yml` file, and therefore need an application name to run. Optionally, you can specify an environment name by adding a `--env` option (dev by default).

Listing 16-27

```
$ php symfony propel:data-load --env=prod --application=frontend
```

This command reads all the YAML fixture files from the `data/fixtures/` directory and inserts the records into the database. By default, it replaces the existing database content, but if you add an `--append` option, the command will not erase the current data.

Listing 16-28

```
$ php symfony propel:data-load --append --application=frontend
```

You can specify another fixture directory in the call. In this case, add a path relative to the project directory.

Listing 16-29

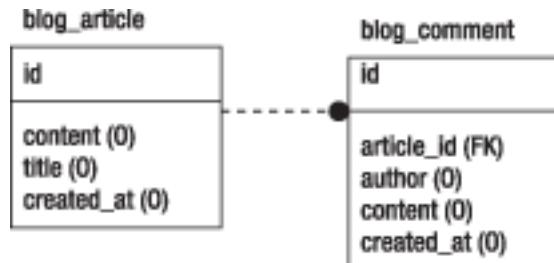
```
$ php symfony propel:data-load --application=frontend data/myfixtures
```

Using Linked Tables

You now know how to add records to a single table, but how do you add records with foreign keys to another table? Since the primary key is not included in the fixtures data, you need an alternative way to relate records to one another.

Let's return to the example in Chapter 8, where a `blog_article` table is linked to a `blog_comment` table, as shown in Figure 16-8.

Figure 16-8 - A sample database relational model



This is where the labels given to the records become really useful. To add a Comment field to the `first_post` article, you simply need to append the lines shown in Listing 16-16 to the `import_data.yml` data file.

Listing 16-16 - Adding a Record to a Related Table, in data/fixtures/import_data.yml

Listing 16-30

```

Comment:
  first_comment:
    article_id: first_post
    author: Anonymous
    content: Your prose is too verbose. Write shorter sentences.
  
```

The `propel:data-load` task will recognize the label that you gave to an article previously in `import_data.yml`, and grab the primary key of the corresponding Article record to set the `article_id` field. You don't even see the IDs of the records; you just link them by their labels—it couldn't be simpler.

The only constraint for linked records is that the objects called in a foreign key must 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 important.

This also works for many-to-many relationships, where two classes are related through a third class. For instance, an `Article` can have many `Authors`, and an `Author` can have many `Articles`. You usually use an `ArticleAuthor` class for that, corresponding to an `article_author` table with an `article_id` and an `author_id` columns. Listing 16-17 shows how to write a fixture file to define many-to-many relationships with this model. Notice the plural table name used here—this is what triggers the search for a middle class.

Listing 16-17 - Adding a Record to a Table Related by a Many-to-Many relationship, in data/fixtures/import_data.yml

```
Author:  
  first_author:  
    name: John Doe  
    article_authors: [first_post, second_post]
```

Listing 16-31

One data file can contain declarations of several classes. But if you need to insert a lot of data for many different tables, your fixture file might get too long to be easily manipulated.

The `propel:data-load` task parses all the files it finds in the `fixtures/` directory, so nothing prevents you from splitting a YAML fixture file into smaller pieces. The important thing to keep in mind is that foreign keys impose a processing order for the tables. To make sure that they are parsed in the correct order, prefix the files with an ordinal number.

```
100_article_import_data.yml  
200_comment_import_data.yml  
300_rating_import_data.yml
```

Listing 16-32



Doctrine does not need specific file names as it automatically takes care of executing the SQL statements in the right order.

Deploying Applications

Symfony offers shorthand commands to synchronize two versions of a website. These commands are mostly used to deploy a website from a development server to a final host, connected to the Internet.

Using rsync for Incremental File Transfer

Sending the root project directory by FTP is fine for the first transfer, but when you need to upload an update of your application, where only a few files have changed, FTP is not ideal. You need to either transfer the whole project again, which is a waste of time and bandwidth, or browse to the directories where you know that some files changed, and transfer only the ones with different modification dates. That's a time-consuming 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 an SSH layer. Rsync⁵⁷ is a command-line utility that provides fast incremental file transfer, and it's open source. With incremental transfer, only the modified data will be sent. If a file didn't change, it won't be sent to the host. If a file changed only partially, just the differential will be sent. The major advantage is that rsync synchronizations transfer only a small amount of data and are very fast.

57. <http://samba.anu.edu.au/rsync/>

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 to avoid security breaches.

The SSH client called by symfony uses connection settings from the `config/properties.ini` file. Listing 16-18 gives an example of connection settings for a production server. Write the settings of your own production server in this file before any synchronization. You can also define a single parameters setting to provide your own rsync command line parameters.

Listing 16-18 - Sample Connection Settings for a Server Synchronization, in myproject/config/properties.ini

```
Listing 16-33 name=myproject

[production]
host=myapp.example.com
port=22
user=myuser
dir=/home/myaccount/myproject/
```



Don't confuse the production server (the host server, as defined in the `properties.ini` file of the project) with the production environment (the front controller and configuration used in production, as referred to in the configuration files of an application).

Doing an 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:

```
Listing 16-34 $ php symfony project:deploy production
```

This command launches the `rsync` command in dry mode; that is, it shows which files must be synchronized but doesn't actually synchronize them. If you want the synchronization to be done, you need to request it explicitly by adding the `--go` option.

```
Listing 16-35 $ php symfony project:deploy production --go
```

Don't forget to clear the cache in the production server after synchronization.



Before deploying your application to the production servers, it is better to first check your configuration with the `check_configuration.php`. This utility can be found in the `data/bin` folder of symfony. It checks your environment against symfony requirements. You can launch it from anywhere:

```
Listing 16-36 $ php /path/to/symfony/data/bin/check_configuration.php
```

Even if you can use this utility from the command line, it's strongly recommended to launch it from the web by copying it under your web root directory as PHP can use different `php.ini` configuration files for the command line interface and the web.

Is your application finished?

Before sending your application to production, you should make sure that it is ready for a public use. Check that the following items are done before actually deciding to deploy the application:

The error pages should be customized to the look and feel of your application. Refer to Chapter 19 to see how to customize the error 500, error 404, and security pages, and to the “Managing a Production Application” section in this chapter to see how to customize the pages displayed when your site is not available.

The `default` module should be removed from the `enabled_modules` array in the `settings.yml`, so that no symfony page appear by mistake.

The session-handling mechanism uses a cookie on the client side, and this cookie is called `symfony` by default. Before deploying your application, you should probably rename it to avoid disclosing the fact that your application uses symfony. Refer to Chapter 6 to see how to customize the cookie name in the `factories.yml` file.

The `robots.txt` file, located in the project’s `web/` directory, is empty by default. You should customize it to inform web spiders and other web robots about which parts of a website they can browse and which they should avoid. Most of the time, this file is used to exclude certain URL spaces from being indexed—for instance, resource-intensive pages, pages that don’t need indexing (such as bug archives), or infinite URL spaces in which robots could get trapped.

Modern browsers request a `favicon.ico` file when a user first browses to your application, to represent the application with an icon in the address bar and bookmarks folder. Providing such a file will not only make your application’s look and feel complete, but it will also prevent a lot of 404 errors from appearing in your server logs.

Ignoring Irrelevant Files

If you synchronize your symfony project with a production host, a few files and directories should not be transferred:

- All the version control directories (`.svn/`, `CVS/`, and so on) and their content are necessary only for development and integration.
- The 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 away from the public.
- 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 way.
- The files uploaded by users should not be transferred. One of the good practices of symfony projects is to store the uploaded files in the `web/uploads/` directory. This allows you to exclude all these files from the synchronization 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 a file, a directory, or a pattern. The symfony file structure is organized logically, and designed to minimize the number of files or directories to exclude manually from the synchronization. See Listing 16-19 for an example.

Listing 16-19 - Sample rsync Exclusion Settings, in myproject/config/rsync_exclude.txt

Listing 16-37

```
# Project files
/cache/*
/log/*
/web/*_dev.php
/web/uploads/*

# SCM files
.arch-params
.bzr
_darcs
.git
.hg
.monotone
.svn
CVS
```



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.

Managing a Production Application

The command that is used most often in production servers is `cache:clear`. You must run it every time you upgrade symfony or your project (for instance, after calling the `project:deploy` task), and every time you change the configuration in production.

Listing 16-38

```
$ php symfony cache:clear
```



If the command-line interface is not available in your production server, you can still clear the cache manually by erasing the contents of the `cache/` folder.

You can temporarily disable your application—for instance, when you need to upgrade a library or a large amount of data.

Listing 16-39

```
$ php symfony project:disable APPLICATION_NAME ENVIRONMENT_NAME
```

By default, a disabled application displays the `sfConfig::get('sf_symfony_lib_dir')/exception/data/unavailable.php` page, but if you create your own `unavailable.php` file in your project's `config/` directory, symfony will use it instead.

The `project:enable` task reenables the application and clears the cache.

Listing 16-40

```
$ php symfony project:enable APPLICATION_NAME ENVIRONMENT_NAME
```



`project:disable` currently has no effect if the `check_lock` parameter is not set to `true` in `settings.yml`.

Displaying an unavailable page when clearing the cache

If you set the `check_lock` parameter to `true` in the `settings.yml` file, symfony will lock the application when the cache is being cleared, and all the requests arriving before the cache is finally cleared are then redirected to a page saying that the application is temporarily unavailable. If the cache is large, the delay to clear it may be longer than a few milliseconds, and if your site's traffic is high, this is a recommended setting. This unavailable page is the same as the one displayed when you call the symfony `disable` task. The `check_lock` parameter is deactivated by default because it has a very slight negative impact on performance.

The `project:clear-controllers` task clears the `web/` directory of all controllers other than the ones running in a production environment. If you do not include the development front controllers in the `rsync_exclude.txt` file, this command guarantees that a backdoor will not reveal the internals of your application.

```
$ php symfony project:clear-controllers
```

*Listing
16-41*

The permissions of the project files and directories can be broken if you use a checkout from an SVN repository. The `project:permissions` task fixes directory permissions, to change the `log/` and `cache/` permissions to `0777`, for example (these directories need to be writable for the framework to work correctly).

```
$ php symfony project:permissions
```

*Listing
16-42*

Summary

By combining PHP logs and symfony logs, you can monitor and debug your application easily. During development, the debug mode, the exceptions, and the web debug toolbar help you locate problems. You can even insert custom messages in the log files or in the toolbar for easier debugging.

The command-line interface provides a large number of tools that facilitate the management of your applications, during development and production phases. Among others, the data population, and synchronization tasks are great time-savers.

Chapter 17

Extending Symfony

Eventually, you will need to alter symfony's behavior. Whether you need to modify the way a certain class behaves or add your own custom features, the moment will inevitably happen because all clients have specific requirements that no framework can forecast. Actually, this situation is so common that symfony provides a mechanism to extend existing classes at runtime, beyond simple class inheritance. You can even replace the core symfony classes by modifying the factories settings. Once you have built an extension, you can easily package it as a plug-in, so that it can be reused in other applications, or by other symfony users.

Events

PHP does not support multiple inheritance, which means it is not possible to have a class extend more than one other class. Also it is not possible to add new methods to an existing class or override existing methods. To ease these two limitations and to make the framework truly extendable, symfony introduces an *event system*, inspired by the Cocoa notification center, and based on the Observer design pattern⁵⁸.

Understanding Events

Some of the symfony classes "notify the dispatcher of an event" at various moments of their life. For instance, when the user changes their culture, the user object notifies that a `change_culture` event has occurred. This is like a shout in the project's space, saying: "I'm doing that. Do whatever you want about it".

You can decide to do something special when an event is fired. For instance, you could save the user culture to a database table each time the `change_culture` event occurs. In order to do so, you need to *register an event listener*, in other words you must declare a function that will be called when the event occurs. Listing 17-1 shows how to register a listener on the user's `change_culture` event.

Listing 17-1 - Registering an Event Listener

```
Listing 17-1 $dispatcher->connect('user.change_culture', 'changeUserCulture');

function changeUserCulture(sfEvent $event)
{
    $user = $event->getSubject();
    $culture = $event['culture'];
```

58. http://en.wikipedia.org/wiki/Observer_pattern

```
// do something with the user culture
}
```

All events and listener registrations are managed by a special object called the *event dispatcher*. This object is available from everywhere in symfony by way of the `ProjectConfiguration` instance, and most symfony objects offer a `getEventDispatcher()` method to get direct access to it. Using the dispatcher's `connect()` method, you can register any PHP callable (either a class method or a function) to be called when an event occurs. The first argument of `connect()` is the event identifier, which is a string composed of a namespace and a name. The second argument is a PHP callable.



Retrieving the event dispatcher from anywhere in the application:

```
$dispatcher = ProjectConfiguration::getActive()->getEventDispatcher();
```

Listing 17-2

Once the function is registered with the event dispatcher, it waits until the event is fired. The event dispatcher keeps a record of all event listeners, and knows which ones to call when an event occurs. When calling these methods or functions, the dispatcher passes them an `sfEvent` object as a parameter.

The event object stores information about the notified event. The event notifier can be retrieved thanks to the `getSubject()` method, and the event parameters are accessible by using the event object as an array (for example, `$event['culture']` can be used to retrieve the `culture` parameter passed by `sfUser` when notifying `user.change_culture`).

To wrap up, the event system allows you to add abilities to an existing class or modify its methods at runtime, without using inheritance.

Notifying an Event listener

Just like symfony classes notify that events have occurred, your own classes can offer runtime extensibility and notify of events at certain occasions. For instance, let's say that your application requests several third-party web services, and that you have written an `sfRestRequest` class to wrap the REST logic of these requests. A good idea would be to trigger an event each time this class makes a new request. This would make the addition of logging or caching capabilities easier in the future. Listing 17-2 shows the code you need to add to an existing `fetch()` method to make it notify an event listener.

Listing 17-2 - Notifying an Event listener

```
class sfRestRequest
{
    protected $dispatcher = null;

    public function __construct(sfEventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    /**
     * Makes a query to an external web service
     */
    public function fetch($uri, $parameters = array())
    {
        // Notify the dispatcher of the beginning of the fetch process
        $this->dispatcher->notify(new sfEvent($this,
```

Listing 17-3

```
'rest_request.fetch_prepare', array(
    'uri'          => $uri,
    'parameters'  => $parameters
));

// Make the request and store the result in a $result variable
// ...

// Notify the dispatcher of the end of the fetch process
$this->dispatcher->notify(new sfEvent($this,
'rest_request.fetch_success', array(
    'uri'          => $uri,
    'parameters'  => $parameters,
    'result'       => $result
)));

return $result;
}
}
```

The `notify()` method of the event dispatcher expects an `sfEvent` object as an argument; this is the very same object that is passed to the event listeners. This object always carries a reference to the notifier (that's why the event instance is initialized with `this`) and an event identifier. Optionally, it accepts an associative array of parameters, giving listeners a way to interact with the notifier's logic.

Notifying the dispatcher of an Event Until a Listener handles it

By using the `notify()` method, you make sure that all the listeners registered on the notifying event are executed. However, in some cases you need to allow a listener to stop the event and prevent further listeners from being notified about it. In this case, you should use `notifyUntil()` instead of `notify()`. The dispatcher will then execute all listeners until one returns `true`, and then stop the event notification. In other words, `notifyUntil()` is like a shout in the project space saying: "I'm doing that. If somebody cares, then I won't tell anybody else". Listing 17-3 shows how to use this technique in combination with a magic `__call()` method to add methods to an existing class at runtime.

Listing 17-3 - Notifying of an Event Until a Listener Returns True

Listing
17-4

```
class sfRestRequest
{
    // ...

    public function __call($method, $arguments)
    {
        $event = $this->dispatcher->notifyUntil(new sfEvent($this,
'rest_request.method_not_found', array(
    'method'      => $method,
    'arguments'   => $arguments
)));
        if (!$event->isProcessed())
        {
            throw new sfException(sprintf('Call to undefined method %s::%s.',
get_class($this), $method));
        }

        return $event->getReturnValue();
    }
}
```

```

    }
}
```

An event listener registered on the `rest_request.method_not_found` event can test the requested `$method` and decide to handle it, or pass to the next event listener callable. In Listing 17-4, you can see how a third party class can add `put()` and `delete()` methods to the `sfRestRequest` class at runtime with this trick.

Listing 17-4 - Handling a “Notify Until” Event type

```

class frontendConfiguration extends sfApplicationConfiguration
{
    public function configure()
    {
        // ...

        // Register our listener
        $this->dispatcher->connect('rest_request.method_not_found',
array('sfRestRequestExtension', 'listenToMethodNotFound'));
    }
}

class sfRestRequestExtension
{
    static public function listenToMethodNotFound(sfEvent $event)
    {
        switch ($event['method'])
        {
            case 'put':
                self::put($event->getSubject(), $event['arguments']);

                return true;
            case 'delete':
                self::delete($event->getSubject(), $event['arguments']);

                return true;
            default:
                return false;
        }
    }

    static protected function put($restRequest, $arguments)
    {
        // Make a put request and store the result in a $result variable
        // ...

        $event->setReturnValue($result);
    }

    static protected function delete($restRequest, $arguments)
    {
        // Make a delete request and store the result in a $result variable
        // ...

        $event->setReturnValue($result);
    }
}
```

*Listing
17-5*

In practice, `notifyUntil()` offers multiple inheritance capabilities, or rather mixins (the addition of methods from third-party classes to an existing class), to PHP. You can now “inject” new methods to objects that you can’t extend by way of inheritance. And this happens at runtime. You are not limited by the Object Oriented capabilities of PHP anymore when you use symfony.



As the first listener to catch a `notifyUntil()` event prevents further notifications, you may worry about the order in which listeners are executed. This order corresponds to the order in which listeners were registered - first registered, first executed. In practice, cases where this could be an issue seldom happen. If you realize that two listeners conflict on a particular event, perhaps your class should notify several events, for instance one at the beginning and one at the end of the method execution. And if you use events to add new methods to an existing class, name your methods wisely so that other attempts at adding methods don’t conflict. Prefixing method names with the name of the listener class is a good practice.

Changing the Return Value of a Method

You can probably imagine how a listener can not only use the information given by an event, but also modify it, to alter the original logic of the notifier. If you want to allow this, you should use the `filter()` method of the event dispatcher rather than `notify()`. All event listeners are then called with two parameters: the event object, and the value to filter. Event listeners must return the value, whether they altered it or not. Listing 17-5 shows how `filter()` can be used to filter a response from a web service and escape special characters in that response.

Listing 17-5 - Notifying of and Handling a Filter Event

```
Listing 17-6
class sfRestRequest
{
    // ...

    /**
     * Make a query to an external web service
     */
    public function fetch($uri, $parameters = array())
    {
        // Make the request and store the result in a $result variable
        // ...

        // Notify of the end of the fetch process
        return $this->dispatcher->filter(new sfEvent($this,
'rest_request.filter_result', array(
    'uri'          => $uri,
    'parameters'  => $parameters,
)), $result)->getReturnValue();
    }
}

// Add escaping to the web service response
$dispatcher->connect('rest_request.filter_result',
'rest_htmlspecialchars');

function rest_htmlspecialchars(sfEvent $event, $result)
{
    return htmlspecialchars($result, ENT_QUOTES, 'UTF-8');
}
```

Built-In Events

Many of symfony's classes have built-in events, allowing you to extend the framework without necessarily changing the classes themselves. Table 17-1 lists these events, together with their types and arguments.

Table 17-1 - Symfony's Events

Event name (Type)	Notifiers	Arguments
application.log (notify)	lot of classes	priority
application.throw_exception (notifyUntil)	sfException	-
autoload.filter_config (filter)	sfAutoloadConfigHandler	-
command.log (notify)	sfCommand* classes	priority
command.pre_command (notifyUntil)	sfTask	arguments, options
command.post_command (notify)	sfTask	-
command.filter_options (filter)	sfTask	command_manager
configuration.method_not_found (notifyUntil)	sfProjectConfiguration	method, arguments
component.method_not_found (notifyUntil)	sfComponent	method, arguments
context.load_factories (notify)	sfContext	-
context.method_not_found (notifyUntil)	sfContext	method, arguments
controller.change_action (notify)	sfController	module, action
controller.method_not_found (notifyUntil)	sfController	method, arguments
controller.page_not_found (notify)	sfController	module, action
debug.web.load_panels (notify)	sfWebDebug	-
debug.web.view.filter_parameter_html (filter)	sfWebDebugPanelView	parameter
doctrine.configure (notify)	sfDoctrinePluginConfiguration	-
doctrine.filter_model_builder_options (filter)	sfDoctrinePluginConfiguration	-
doctrine.filter_cli_config (filter)	sfDoctrinePluginConfiguration	-
doctrine.configure_connection (notify)	Doctrine_Manager	connection, database
doctrine.admin.delete_object (notify)	-	object
doctrine.admin.save_object (notify)	-	object
doctrine.admin.build_query (filter)	-	
doctrine.admin.pre_execute (notify)	-	configuration
form.post_configure (notify)	sfFormSymfony	-
form.filter_values (filter)	sfFormSymfony	-
form.validation_error (notify)	sfFormSymfony	error
form.method_not_found (notifyUntil)	sfFormSymfony	method, arguments
mailer.configure (notify)	sfMailer	-

Event name (Type)	Notifiers	Arguments
plugin.pre_install (notify)	sfPluginManager	channel, plugin, is_package
plugin.post_install (notify)	sfPluginManager	channel, plugin
plugin.pre_uninstall (notify)	sfPluginManager	channel, plugin
plugin.post_uninstall (notify)	sfPluginManager	channel, plugin
propel.configure (notify)	sfPropelPluginConfiguration	-
propel.filter_phing_args (filter)	sfPropelBaseTask	-
propel.filter_connection_config (filter)	sfPropelDatabase	name, database
propel.admin.delete_object (notify)	-	object
propel.admin.save_object (notify)	-	object
propel.admin.build_criteria (filter)	-	
propel.admin.pre_execute (notify)	-	configuration
request.filter_parameters (filter)	sfWebRequest	path_info
request.method_not_found (notifyUntil)	sfRequest	method, arguments
response.method_not_found (notifyUntil)	sfResponse	method, arguments
response.filter_content (filter)	sfResponse, sfException	-
routing.load_configuration (notify)	sfRouting	-
task.cache.clear (notifyUntil)	sfCacheClearTask	app, type, env
task.test.filter_test_files (filter)	sfTestBaseTask	arguments, options
template.filter_parameters (filter)	sfViewParameterHolder	-
user.change_culture (notify)	sfUser	culture
user.method_not_found (notifyUntil)	sfUser	method, arguments
user.change_authentication (notify)	sfBasicSecurityUser	authenticated
view.configure_format (notify)	sfView	format, response, request
view.method_not_found (notifyUntil)	sfView	method, arguments
view.cache.filter_content (filter)	sfViewCacheManager	response, uri, new

You are free to register event listeners on any of these events. Just make sure that listener callables return a Boolean when registered on a `notifyUntil` event type, and that they return the filtered value when registered on a `filter` event type.

Note that the event namespaces don't necessarily match the class role. For instance, all symfony classes notify of an `application.log` event when they need something to appear in the log files (and in the web debug toolbar):

Listing 17-7

```
$dispatcher->notify(new sfEvent($this, 'application.log',
array($message)));
```

Your own classes can do the same and also notify symfony events when it makes sense to do so.

Where To Register Listeners?

Event listeners need to be registered early in the life of a symfony request. In practice, the right place to register event listeners is in the application configuration class. This class has a reference to the event dispatcher that you can use in the `configure()` method. Listing 17-6 shows how to register a listener on one of the `rest_request` events of the above examples.

Listing 17-6 - Registering a Listener in the Application Configuration Class, in `apps/frontend/config/ApplicationConfiguration.class.php`

```
class frontendConfiguration extends sfApplicationConfiguration
{
    public function configure()
    {
        // ...

        $this->dispatcher->connect('rest_request.method_not_found',
array('sfRestRequestExtension', 'listenToMethodNotFound'));
    }
}
```

*Listing
17-8*

Plug-ins (see below) can register their own event listeners. They should do it in the plug-in's `config/config.php` script, which is executed during application initialization and offers access to the event dispatcher through `$this->dispatcher`.

Factories

A factory is the definition of a class for a certain task. Symfony relies on factories for its core features such as the controller and session capabilities. For instance, when the framework needs to create a new request object, it searches in the factory definition for the name of the class to use for that purpose. The default factory definition for requests is `sfWebRequest`, so symfony creates an object of this class in order to deal with requests. The great advantage of using a factory definition is that it is very easy to alter the core features of the framework: Just change the factory definition, and symfony will use your custom request class instead of its own.

The factory definitions are stored in the `factories.yml` configuration file. Listing 17-7 shows the default factory definition file. Each definition is made of the name of an autoloaded class and (optionally) a set of parameters. For instance, the session storage factory (set under the `storage:` key) uses a `session_name` parameter to name the cookie created on the client computer to allow persistent sessions.

Listing 17-7 - Default Factories File, in `frontend/config/factories.yml`

```
-  
prod:  
  logger:  
    class:  sfNoLogger  
    param:  
      level:  err  
    loggers: ~  
  
test:  
  storage:  
    class: sfSessionTestStorage  
    param:  
      session_path: %SF_TEST_CACHE_DIR%/sessions
```

*Listing
17-9*

```

response:
    class: sfWebResponse
    param:
        send_http_headers: false

mailer:
    param:
        delivery_strategy: none

dev:
    mailer:
        param:
            delivery_strategy: none

all:
    routing:
        class: sfPatternRouting
        param:
            generate_shortest_url: true
            extra_parameters_as_query_string: true

    view_cache_manager:
        class: sfViewCacheManager
        param:
            cache_key_use_vary_headers: true
            cache_key_use_host_name: true

```

The best way to change a factory is to create a new class inheriting from the default factory and to add new methods to it. For instance, the user session factory is set to the `myUser` class (located in `frontend/lib/`) and inherits from `sfUser`. Use the same mechanism to take advantage of the existing factories. Listing 17-8 shows an example of a new factory for the request object.

Listing 17-8 - Overriding Factories

```

Listing 17-10 // Create a myRequest.class.php in an autoloaded directory,
// For instance in frontend/lib/
<?php

class myRequest extends sfRequest
{
    // Your code here
}

// Declare this class as the request factory in factories.yml
all:
    request:
        class: myRequest

```

Plug-Ins

You will probably need to reuse a piece of code that you developed for one of your symfony applications. If you can package this piece of code into a single class, no problem: Drop the class in one of the `lib/` folders of another application and the autoloader will take care of the rest. But if the code is spread across more than one file, such as a complete new theme for

the administration generator or a combination of JavaScript files and helpers to automate your favorite visual effect, just copying the files is not the best solution.

Plug-ins offer a way to package code disseminated in several files and to reuse this code across several projects. Into a plug-in, you can package classes, filters, event listeners, helpers, configuration, tasks, modules, schemas and model extensions, fixtures, web assets, etc. Plug-ins are easy to install, upgrade, and uninstall. They can be distributed as a .tgz archive, a PEAR package, or a simple checkout of a code repository. The PEAR packaged plug-ins have the advantage of managing dependencies, being easier to upgrade and automatically discovered. The symfony loading mechanisms take plug-ins into account, and the features offered by a plug-in are available in the project as if the plug-in code was part of the framework.

So, basically, a plug-in is a packaged extension for a symfony project. With plug-ins, not only can you reuse your own code across applications, but you can also reuse developments made by other contributors and add third-party extensions to the symfony core.

Finding Symfony Plug-Ins

The symfony project website contains a section dedicated to symfony plug-ins and accessible with the following URL:

<http://www.symfony-project.org/plugins/>

*Listing
17-11*

Each plug-in listed there has its own page, with detailed installation instructions and documentation.

Some of these plug-ins are contributions from the community, and some come from the core symfony developers. Among the latter, you will find the following:

- **sfFeed2Plugin**: Automates the manipulation of RSS and Atom feeds
- **sfThumbnailPlugin**: Creates thumbnails—for instance, for uploaded images
- **sfMediaLibraryPlugin**: Allows media upload and management, including an extension for rich text editors to allow authoring of images inside rich text
- **sfGuardPlugin**: Provides authentication, authorization, and other user management features above the standard security feature of symfony
- **sfSuperCachePlugin**: Writes pages in cache directory under the web root to allow the web server to serve them as fast as possible
- **sfErrorLoggerPlugin**: Logs every 404 and 500 error in a database and provides an administration module to browse these errors
- **sfSslRequirementPlugin**: Provides SSL encryption support for actions

You should regularly check out the symfony plugin section, because new plug-ins are added all the time, and they bring very useful shortcuts to many aspects of web application programming.

Apart from the symfony plugin section, the other ways to distribute plug-ins are to propose a plug-ins archive for download, to host them in a PEAR channel, or to store them in a public version control repository.

Installing a Plug-In

The plug-in installation process differs according to the way it's packaged. Always refer to the included README file and/or installation instructions on the plug-in download page.

Plug-ins are installed applications on a per-project basis. All the methods described in the following sections result in putting all the files of a plug-in into a `myproject/plugins/pluginName/` directory.

PEAR Plug-Ins

Plug-ins listed on the symfony plugin section can be bundled as PEAR packages and made available via the official symfony plugins PEAR channel: `plugins.symfony-project.org`. To install such a plug-in, use the `plugin:install` task with a plugin name, as shown in Listing 17-9.

Listing 17-9 - Installing a Plug-In from the Official symfony plugins PEAR Channel

```
Listing 17-12 $ cd myproject
$ php symfony plugin:install pluginName
```

Alternatively, you can download the plug-in and install it from the disk. In this case, use the path to the package archive, as shown in Listing 17-10.

Listing 17-10 - Installing a Plug-In from a Downloaded PEAR Package

```
Listing 17-13 $ cd myproject
$ php symfony plugin:install /home/path/to/downloads/pluginName.tgz
```

Some plug-ins are hosted on external PEAR channels. Install them with the `plugin:install` task, and don't forget to register the channel and mention the channel name, as shown in Listing 17-11.

Listing 17-11 - Installing a Plug-In from a PEAR Channel

```
Listing 17-14 $ cd myproject
$ php symfony plugin:add-channel channel.symfony.pear.example.com
$ php symfony plugin:install --channel=channel.symfony.pear.example.com
pluginName
```

These three types of installation all use a PEAR package, so the term “PEAR plug-in” will be used indiscriminately to talk about plug-ins installed from the symfony plugins PEAR channel, an external PEAR channel, or a downloaded PEAR package.

The `plugin:install` task also takes a number of options, as shown on Listing 17-12.

Listing 17-12 - Installing a Plug-In with some Options

```
Listing 17-15 $ php symfony plugin:install --stability=beta pluginName
$ php symfony plugin:install --release=1.0.3 pluginName
$ php symfony plugin:install --install-deps pluginName
```



As for every symfony task, you can have a full explanation of the `plugin:install` options and arguments by launching `php symfony help plugin:install`.

Archive Plug-Ins

Some plug-ins come as a simple archive of files. To install those, just unpack the archive into your project's `plugins/` directory. If the plug-in contains a `web/` subdirectory, don't forget to run the `plugin:publish-assets` command to create the corresponding symlink under the main `web/` folder as shown in listing 17-13. Finally, don't forget to clear the cache.

Listing 17-13 - Installing a Plug-In from an Archive

```
Listing 17-16 $ cd plugins
$ tar -zxf myPlugin.tgz
$ cd ..
$ php symfony plugin:publish-assets
$ php symfony cc
```

Installing Plug-Ins from a Version Control Repository

Plug-ins sometimes have their own source code repository for version control. You can install them by doing a simple checkout in the `plugins/` directory, but this can be problematic if your project itself is under version control.

Alternatively, you can declare the plug-in as an external dependency so that every update of your project source code also updates the plug-in source code. For instance, Subversion stores external dependencies in the `svn:externals` property. So you can add a plug-in by editing this property and updating your source code afterwards, as Listing 17-14 demonstrates.

Listing 17-14 - Installing a Plug-In from a Source Version Repository

```
$ cd myproject
$ svn propedit svn:externals plugins
  pluginName  http://svn.example.com/pluginName/trunk
$ svn up
$ php symfony plugin:publish-assets
$ php symfony cc
```

*Listing
17-17*



If the plug-in contains a `web/` directory, the `symfony plugin:publish-assets` command has to be run to generate the corresponding symlink under the main `web/` folder of the project.

Activating a Plug-In Module

Some plug-ins contain whole modules. The only difference between module plug-ins and classical modules is that module plug-ins don't appear in the `myproject/apps/frontend/modules/` directory (to keep them easily upgradeable). They also need to be activated in the `settings.yml` file, as shown in Listing 17-15.

Listing 17-15 - Activating a Plug-In Module, in frontend/config/settings.yml

```
all:
  .settings:
    enabled_modules: [default, sfMyPluginModule]
```

*Listing
17-18*

This is to avoid a situation where the plug-in module is mistakenly made available for an application that doesn't require it, which could open a security breach. Think about a plug-in that provides `frontend` and `backend` modules. You will need to enable the `frontend` modules only in your `frontend` application, and the `backend` ones only in the `backend` application. This is why plug-in modules are not activated by default.



The default module is the only enabled module by default. That's not really a plug-in module, because it resides in the framework, in `sfConfig::get('sf_symfony_lib_dir')/controller/default/`. This is the module that provides the congratulations pages, and the default error pages for 404 and credentials required errors. If you don't want to use the symfony default pages, just remove this module from the `enabled_modules` setting.

Listing the Installed Plug-Ins

If a glance at your project's `plugins/` directory can tell you which plug-ins are installed, the `plugin:list` task tells you even more: the version number and the channel name of each installed plug-in (see Listing 17-16).

Listing 17-16 - Listing Installed Plug-Ins

```
Listing 17-19 $ cd myproject
$ php symfony plugin:list

Installed plugins:
sfPrototypePlugin          1.0.0-stable # plugins.symfony-project.com
(sf Symfony)
sfSuperCachePlugin          1.0.0-stable # plugins.symfony-project.com
(sf Symfony)
sfThumbnail                 1.1.0-stable # plugins.symfony-project.com
(sf Symfony)
```

Upgrading and Uninstalling Plug-Ins

To uninstall a PEAR plug-in, call the `plugin:uninstall` task from the root project directory, as shown in Listing 17-17. You must prefix the plug-in name with its installation channel if it's different from the default `symfony` channel (use the `plugin:list` task to determine this channel).

Listing 17-17 - Uninstalling a Plug-In

```
Listing 17-20 $ cd myproject
$ php symfony plugin:uninstall sfSuperCachePlugin
$ php symfony cc
```

To uninstall an archive plug-in or an SVN plug-in, remove manually the plug-in files from the project `plugins/` and `web/` directories, and clear the cache.

To upgrade a plug-in, either use the `plugin:upgrade` task (for a PEAR plug-in) or do an `svn update` (if you grabbed the plug-in from a version control repository). Archive plug-ins can't be upgraded easily.

Anatomy of a Plug-In

Plug-ins are written using the PHP language. If you can understand how an application is organized, you can understand the structure of the plug-ins.

Plug-In File Structure

A plug-in directory is organized more or less like a project directory. The plug-in files have to be in the right directories in order to be loaded automatically by `symfony` when needed. Have a look at the plug-in file structure description in Listing 17-18.

Listing 17-18 - File Structure of a Plug-In

```
Listing 17-21 pluginName/
  config/
    routing.yml      // Routing config file
    app.yml         // Plugin default settings
    *schema.yml     // Data schema
    *schema.xml
    config.php      // Specific plug-in configuration
  data/
    generator/
      sfPropelAdmin
      */
      template/
      skeleton/
  fixtures/
    *.yml          // Fixtures files
```

```

lib/
  *.php          // Classes
  helper/
    *.php        // Helpers
  model/
    *.php        // Model classes
  task/
    *Task.class.php // CLI tasks
modules/
  /*             // Modules
  actions/
    actions.class.php
  config/
    module.yml
    view.yml
    security.yml
  templates/
    *.php
web/
  *              // Assets

```

Plug-In Abilities

Plug-ins can contain a lot of things. Their content is automatically taken into account by your application at runtime and when calling tasks with the command line. But for plug-ins to work properly, you must respect a few conventions:

- Database schemas are detected by the `propel-` tasks. When you call `propel:build --classes` or `doctrine:build --classes` in your project, you rebuild the project model and all the plug-in models with it. Note that a Propel plug-in schema must always have a `package` attribute under the shape `plugins.pluginName.lib.model`, as shown in Listing 17-19. If you use Doctrine, the task will automatically generate the classes in the plugin directory.

Listing 17-19 - Example of Propel Schema Declaration in a Plug-In, in myPlugin/config/schema.yml

```

propel:
  _attributes: { package: plugins.myPlugin.lib.model }
  my_plugin_foo:
    _attributes: { phpName: myPluginFoo }
    id:
      name: { type: varchar, size: 255, index: unique }
    ...

```

*Listing
17-22*

- The plug-in configuration is to be included in the plug-in configuration class (`PluginNameConfiguration.class.php`). This file is executed after the application and project configuration, so `symfony` is already bootstrapped at that time. You can use this file, for instance, to extend existing classes with event listeners and behaviors.
- Fixtures files located in the plug-in `data/fixtures/` directory are processed by the `propel:data-load` or `doctrine:data-load` task.
- Custom classes are autoloaded just like the ones you put in your project `lib/` folders.
- Helpers are automatically found when you call `use_helper()` in templates. They must be in `a helper/` subdirectory of one of the plug-in's `lib/` directory.

- If you use Propel, model classes in `myplugin/lib/model/` specialize the model classes generated by the Propel builder (in `myplugin/lib/model/om/` and `myplugin/lib/model/map/`). They are, of course, autoloaded. Be aware that you cannot override the generated model classes of a plug-in in your own project directories.
- If you use Doctrine, the ORM generates the plugins base classes in `myplugin/lib/model/Plugin*.class.php`, and concrete classes in `lib/model/myplugin/`. This means that you can easily override the model classes in your application.
- Tasks are immediately available to the symfony command line as soon as the plug-in is installed. A plugin can either add new tasks, or override an existing one. It is a best practice to use the plug-in name as a namespace for the task. Type `php symfony` to see the list of available tasks, including the ones added by plug-ins.
- Modules provide new actions accessible from the outside, provided that you declare them in the `enabled_modules` setting in your application.
- Web assets (images, scripts, style sheets, etc.) are made available to the server. When you install a plug-in via the command line, `symfony` creates a symlink to the project `web/` directory if the system allows it, or copies the content of the module `web/` directory into the project one. If the plug-in is installed from an archive or a version control repository, you have to copy the plug-in `web/` directory by hand (as the `README` bundled with the plug-in should mention).



Registering routing rules in a Plug-in A plug-in can add new rules to the routing system, but it is not recommandable to do it by using a custom `routing.yml` configuration file. This is because the order in which rules are defined is very important, and the simple cascade configuration system of YAML files in `symfony` would mess this order up. Instead, plug-ins need to register an event listener on the `routing.load_configuration` event and manually prepend rules in the listener:

```
Listing 17-23 // in plugins/myPlugin/config/config.php
$this->dispatcher->connect('routing.load_configuration',
array('myPluginRouting', 'listenToRoutingLoadConfigurationEvent'));

// in plugins/myPlugin/lib/myPluginRouting.php
class myPluginRouting
{
    static public function listenToRoutingLoadConfigurationEvent(sfEvent
$event)
    {
        $routing = $event->getSubject();
        // add plug-in routing rules on top of the existing ones
        $routing->prependRoute('my_route', new sfRoute('/my_plugin/:action',
array('module' => 'myPluginAdministrationInterface')));
    }
}
```

TIP: Defining the default settings in `config/app.yml` The plugin own default settings can be defined in its `config/app.yml` file. However overriding settings defined in an other plugin is not safe as the final value would depends on the `app.yml` files loading order. Custom application configuration can be used in the plug-in code (for instance, by using `sfConfig::get('app_myplugin_foo')`) and the settings can be overridden at the application level (see Listing 17-20 for an example) Handling of the default values can be done by using the second argument of the `sfConfig::get()` method or by defining it in the plugin `app.yml` file.

Manual Plug-In Setup

There are some elements that the `plugin:install` task cannot handle on its own, and which require manual setup during installation:

- Custom routing should be added either by the plugin code on the `routing.load_configuration` event or manually to the application `routing.yml`.
- Custom filters have to be added manually to the application `filters.yml`.
- Custom factories have to be added manually to the application `factories.yml`.

Plug-ins with such manual setup should embed a `README` file describing installation in detail.

Customizing a Plug-In for an Application

Whenever you want to customize a plug-in, never alter the code found in the `plugins/` directory. If you do so, you will lose all your modifications when you upgrade the plug-in. For customization needs, plug-ins provide custom settings, and they support overriding.

Well-designed plug-ins use settings that can be changed in the application `app.yml`, as Listing 17-20 demonstrates.

Listing 17-20 - Customizing a Plug-In That Uses the Application Configuration

```
// example plug-in code
$foo = sfConfig::get('app_my_plugin_foo', 'bar');

// Change the 'foo' default value ('bar') in the application app.yml
all:
  my_plugin:
    foo:      barbar
```

*Listing
17-24*

The module settings and their default values are often described in the plug-in's `README` file. You can replace the default contents of a plug-in module by creating a module of the same name in your own application. It is not really overriding, since the elements in your application are used instead of the ones of the plug-in. It works fine if you create templates and configuration files of the same name as the ones of the plug-ins.

On the other hand, if a plug-in wants to offer a module with the ability to override its actions, the `actions.class.php` in the plug-in module must be empty and inherit from an autoloading class, so that the method of this class can be inherited as well by the `actions.class.php` of the application module. See Listing 17-21 for an example.

Listing 17-21 - Customizing a Plug-In Action

```
// In myPlugin/modules/mymodule/lib/myPluginmymoduleActions.class.php
class myPluginmymoduleActions extends sfActions
{
  public function executeIndex()
  {
    // Some code there
  }
}

// In myPlugin/modules/mymodule/actions/actions.class.php
require_once dirname(__FILE__).'/../lib/myPluginmymoduleActions.class.php';

class mymoduleActions extends myPluginmymoduleActions
{
```

*Listing
17-25*

```
// Nothing
}

// In frontend/modules/mymodule/actions/actions.class.php
class mymoduleActions extends myPluginmymoduleActions
{
    public function executeIndex()
    {
        // Override the plug-in code there
    }
}
```

Customizing the plug-in schema

Doctrine

When building the model, Doctrine will look for all the `*schema.yml` in application's and plugins' `config/` directories, so a project schema can override a plugin schema. The merging process allows for addition or modification of table or columns. For instance, the following example shows how to add columns to a table defined in a plugin schema.

```
#Original schema, in plugins/myPlugin/config/schema.yml
Article:
    columns:
        name: string(50)

#Project schema, in config/schema.yml
Article:
    columns:
        stripped_title: string(50)

# Resulting schema, merged internally and used for model and sql
generation
Article:
    columns:
        name: string(50)
        stripped_title: string(50)
```

*Listing
17-26*

Propel

When building the model, symfony will look for custom YAML files for each existing schema, including plug-in ones, following this rule:

Original schema name	Custom schema name
<code>config/schema.yml</code>	<code>schema.custom.yml</code>
<code>config/foobar_schema.yml</code>	<code>foobar_schema.custom.yml</code>
<code>plugins/myPlugin/config/schema.yml</code>	<code>myPlugin-schema.custom.yml</code>
<code>plugins/myPlugin/config/foo_schema.yml</code>	<code>myPlugin_foo-schema.custom.yml</code>

Custom schemas will be looked for in the application's and plugins' `config/` directories, so a plugin can override another plugin's schema, and there can be more than one customization per schema.

Symfony will then merge the two schemas based on each table's `phpName`. The merging process allows for addition or modification of tables, columns, and column attributes. For instance, the next listing shows how a custom schema can add columns to a table defined in a plug-in schema.

```
# Original schema, in plugins/myPlugin/config/schema.yml
propel:
    article:
        _attributes: { phpName: Article }
        title: varchar(50)
        user_id: { type: integer }
        created_at:

# Custom schema, in myPlugin_schema.custom.yml
```

*Listing
17-27*

```

propel:
  article:
    _attributes: { phpName: Article, package: foo.bar.lib.model }
    stripped_title: varchar(50)

# Resulting schema, merged internally and used for model and sql
generation
propel:
  article:
    _attributes: { phpName: Article, package: foo.bar.lib.model }
    title: varchar(50)
    user_id: { type: integer }
    created_at:
    stripped_title: varchar(50)

```

As the merging process uses the table's `phpName` as a key, you can even change the name of a plugin table in the database, provided that you keep the same `phpName` in the schema.

How to Write a Plug-In

Only plug-ins packaged as PEAR packages can be installed with the `plugin:install` task. Remember that such plug-ins can be distributed via the `symfony` plugin section, a PEAR channel, or a simple file download. So if you want to author a plug-in, it is better to publish it as a PEAR package than as a simple archive. In addition, PEAR packaged plug-ins are easier to upgrade, can declare dependencies, and automatically deploy assets in the `web/` directory.

File Organization

Suppose you have developed a new feature and want to package it as a plug-in. The first step is to organize the files logically so that the `symfony` loading mechanisms can find them when needed. For that purpose, you have to follow the structure given in Listing 17-18. Listing 17-22 shows an example of file structure for an `sfSamplePlugin` plug-in.

Listing 17-22 - Example List of Files to Package As a Plug-In

Listing 17-22

```

sfSamplePlugin/
  README
  LICENSE
  config/
    schema.yml
    sfSamplePluginConfiguration.class.php
  data/
    fixtures/
      fixtures.yml
  lib/
    model/
      sfSampleFooBar.php
      sfSampleFooBarPeer.php
    task/
      sfSampleTask.class.php
    validator/
      sfSampleValidator.class.php
  modules/
    sfSampleModule/
      actions/

```

```

actions.class.php
config/
    security.yml
lib/
    BasesfSampleModuleActions.class.php
templates/
    indexSuccess.php
web/
    css/
        sfSampleStyle.css
    images/
        sfSampleImage.png

```

For authoring, the location of the plug-in directory (`sfSamplePlugin/` in Listing 17-22) is not important. It can be anywhere on the disk.



Take examples of the existing plug-ins and, for your first attempts at creating a plug-in, try to reproduce their naming conventions and file structure.

Creating the package.xml File

The next step of plug-in authoring is to add a `package.xml` file at the root of the plug-in directory. The `package.xml` follows the PEAR syntax. Have a look at a typical symfony plug-in `package.xml` in Listing 17-23.

Listing 17-23 - Example package.xml for a Symfony Plug-In

```

<?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:tasks="http://pear.php.net/dtd/tasks-1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0 http://pear.php.net/
dtd/tasks-1.0.xsd http://pear.php.net/dtd/package-2.0 http://pear.php.net/
dtd/package-2.0.xsd">
    <name>sfSamplePlugin</name>
    <channel>plugins.symfony-project.org</channel>
    <summary>symfony sample plugin</summary>
    <description>Just a sample plugin to illustrate PEAR
packaging</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.org/license">MIT
license</license>

```

*Listing
17-23*

```
<notes>-</notes>
<contents>
<dir name="/">
  <file role="data" name="README" />
  <file role="data" name="LICENSE" />
  <dir name="config">
    <!-- model -->
    <file role="data" name="schema.yml" />
    <file role="data" name="ProjectConfiguration.class.php" />
  </dir>
  <dir name="data">
    <dir name="fixtures">
      <!-- fixtures -->
      <file role="data" name="fixtures.yml" />
    </dir>
  </dir>
  <dir name="lib">
    <dir name="model">
      <!-- model classes -->
      <file role="data" name="sfSampleFooBar.php" />
      <file role="data" name="sfSampleFooBarPeer.php" />
    </dir>
    <dir name="task">
      <!-- tasks -->
      <file role="data" name="sfSampleTask.class.php" />
    </dir>
    <dir name="validator">
      <!-- validators -->
      <file role="data" name="sfSampleValidator.class.php" />
    </dir>
  </dir>
  <dir name="modules">
    <dir name="sfSampleModule">
      <file role="data" name="actions/actions.class.php" />
      <file role="data" name="config/security.yml" />
      <file role="data" name="lib/BasesfSampleModuleActions.class.php" />
      <file role="data" name="templates/indexSuccess.php" />
    </dir>
  </dir>
  <dir name="web">
    <dir name="css">
      <!-- stylesheets -->
      <file role="data" name="sfSampleStyle.css" />
    </dir>
    <dir name="images">
      <!-- images -->
      <file role="data" name="sfSampleImage.png" />
    </dir>
  </dir>
</contents>
<dependencies>
  <required>
    <php>
      <min>5.2.4</min>
    </php>
    <pearinstaller>
      <min>1.4.1</min>
```

```

</pearinstaller>
<package>
  <name>symfony</name>
  <channel>pear.symfony-project.com</channel>
  <min>1.3.0</min>
  <max>1.5.0</max>
  <exclude>1.5.0</exclude>
</package>
</required>
</dependencies>
<phprelease />
<changelog />
</package>

```

The interesting parts here are the `<contents>` and the `<dependencies>` tags, described next. For the rest of the tags, there is nothing specific to symfony, so you can refer to the PEAR online manual⁵⁹ for more details about the `package.xml` format.

Contents

The `<contents>` tag is the place where you must describe the plug-in file structure. This will tell PEAR which files to copy and where. Describe the file structure with `<dir>` and `<file>` tags. All `<file>` tags must have a `role="data"` attribute. The `<contents>` part of Listing 17-23 describes the exact directory structure of Listing 17-22.



The use of `<dir>` tags is not compulsory, since you can use relative paths as `name` values in the `<file>` tags. However, it is recommended so that the `package.xml` file remains readable.

Plug-In Dependencies

Plug-ins are designed to work with a given set of versions of PHP, PEAR, symfony, PEAR packages, or other 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.

You should always declare dependencies on PHP, PEAR, and symfony, at least the ones corresponding to your own installation, as a minimum requirement. If you don't know what to put, add a requirement for PHP 5.2.4, PEAR 1.4, and symfony 1.3.

It is also recommended to add a maximum version number of symfony for each plug-in. This will cause an error message when trying to use a plug-in with a more advanced version of the framework, and this will oblige the plug-in author to make sure that the plug-in works correctly with this version before releasing it again. It is better to have an alert and to download an upgrade rather than have a plug-in fail silently.

If you specify plugins as dependencies, users will be able to install your plugin and all its dependencies with a single command:

```
$ php symfony plugin:install --install-deps sfSamplePlugin
```

Listing 17-30

Building the Plug-In

The PEAR component has a command (`pear package`) that creates the `.tgz` archive of the package, provided you call the command shown in Listing 17-24 from a directory containing a `package.xml`.

59. <http://pear.php.net/manual/en/>

Listing 17-24 - Packaging a Plug-In As a PEAR Package

Listing 17-31

```
$ cd sfSamplePlugin
$ pear package

Package sfSamplePlugin-1.0.0.tgz done
```

Once your plug-in is built, check that it works by installing it yourself, as shown in Listing 17-25.

Listing 17-25 - Installing the Plug-In

Listing 17-32

```
$ cp sfSamplePlugin-1.0.0.tgz /home/production/myproject/
$ cd /home/production/myproject/
$ php symfony plugin:install sfSamplePlugin-1.0.0.tgz
```

According to their description in the <contents> tag, the packaged files will end up in different directories of your project. Listing 17-26 shows where the files of the sfSamplePlugin should end up after installation.

Listing 17-26 - The Plug-In Files Are Installed on the plugins/ and web/ Directories

Listing 17-33

```
plugins/
sfSamplePlugin/
  README
  LICENSE
  config/
    schema.yml
    sfSamplePluginConfiguration.class.php
  data/
    fixtures/
      fixtures.yml
  lib/
    model/
      sfSampleFooBar.php
      sfSampleFooBarPeer.php
    task/
      sfSampleTask.class.php
    validator/
      sfSampleValidator.class.php
  modules/
    sfSampleModule/
      actions/
        actions.class.php
      config/
        security.yml
    lib/
      BasesfSampleModuleActions.class.php
      templates/
        indexSuccess.php
  web/
    sfSamplePlugin/          ## Copy or symlink, depending on system
    css/
      sfSampleStyle.css
    images/
      sfSampleImage.png
```

Test the way the plug-in behaves in your application. If it works well, you are ready to distribute it across projects—or to contribute it to the symfony community.

Hosting Your Plug-In in the Symfony Project Website

A symfony plug-in gets the broadest audience when distributed by the `symfony-project.org` website. Even your own plug-ins can be distributed this way, provided that you follow these steps:

1. Make sure the `README` file describes the way to install and use your plug-in, and that the `LICENSE` file gives the license details. Format your `README` with the Markdown⁶⁰ Formatting syntax.
2. Create a symfony account (<http://www.symfony-project.org/user/new>) and create the plugin (<http://www.symfony-project.org/plugins/new>).
3. Create a PEAR package for your plug-in by calling the `pear package` command, and test it. The PEAR package must be named `sfSamplePlugin-1.0.0.tgz` (1.0.0 is the plug-in version).
4. Upload your PEAR package (`sfSamplePlugin-1.0.0.tgz`).
5. Your plugin must now appear in the list of plugins⁶¹.

If you follow this procedure, users will be able to install your plug-in by simply typing the following command in a project directory:

```
$ php symfony plugin:install sfSamplePlugin
```

*Listing
17-34*

Naming Conventions

To keep the `plugins/` directory clean, ensure all the plug-in names are in camelCase and end with `Plugin` (for example, `shoppingCartPlugin`, `feedPlugin`, and so on). Before naming your plug-in, check that there is no existing plug-in with the same name.



Plug-ins relying on Propel should contain `Propel` in the name (the same goes for if you use Doctrine). For instance, an authentication plug-in using the Propel data access objects should be called `sfPropelAuth`.

Plug-ins should always 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, purpose of the plug-in, its effect, installation and configuration instructions, etc.

Summary

The symfony classes notify events that give them the ability to be modified at the application level. The event mechanism allows multiple inheritance and class overriding at runtime even if the PHP limitations forbid it. So you can easily extend the symfony features, even if you have to modify the core classes for that—the factories configuration is here for that.

Many such extensions already exist; they are packaged as plug-ins, to be easily installed, upgraded, and uninstalled through the symfony command line. Creating a plug-in is as easy as creating a PEAR package, and provides reusability across applications.

The symfony plugin section contains many plug-ins, and you can even add your own. So now that you know how to do it, we hope that you will enhance the symfony core with a lot of useful extensions!

60. <http://daringfireball.net/projects/markdown/syntax>

61. <http://www.symfony-project.org/plugins/>

Chapter 18

Performance

If you expect your website will attract a crowd, performance and optimization issues should be a major factor during the development phase. Rest assured, performance has always been a chief concern among the core symfony developers.

While the advantages gained by accelerating the development process result in some overhead, the core symfony developers have always been cognizant of performance requirements. Accordingly, every class and every method have been closely inspected and optimized to be as fast as possible. The basic overhead, which you can measure by comparing the time to display a “hello, world” message with and without symfony, is minimal. As a result, the framework is scalable and reacts well to stress tests. And as the ultimate proof, some websites with extremely⁶² high⁶³ traffic⁶⁴ (that is, websites with millions of active subscribers and a lot of server-pressuring Ajax interactions) use symfony and are very satisfied with its performance.

But, of course, high-traffic websites often have the means to expand the server farm and upgrade hardware as they see fit. If you don’t have the resources to do this, or if you want to be sure the full power of the framework is always at your disposal, there are a few tweaks that you can use to further speed up your symfony application. This chapter lists some of the recommended performance optimizations at all levels of the framework and they are mostly for advanced users. Some of them were already mentioned throughout the previous chapters, but you will find it useful to have them all in one place.

Tweaking the Server

A well-optimized application should rely on a well-optimized server. You should know the basics of server performance to make sure there is no bottleneck outside symfony. Here are a few things to check to make sure that your server isn’t unnecessarily slow.

Having `magic_quotes_gpc` turned `true` in the `php.ini` slows down an application, because it tells PHP to escape all quotes in request parameters, but symfony will systematically unescape them afterwards, and the only consequence will be a loss of time—and quotes-escaping problems on some platforms. Therefore, turn this setting off if you have access to the PHP configuration.

The more recent PHP release you use, the better (PHP 5.3 is faster than PHP 5.2). So make sure you upgrade your PHP version to benefit from the latest performance improvements.

62. <http://sf-to.org/answers>

63. <http://sf-to.org/delicious>

64. <http://sf-to.org/dailymotion>

The use of a PHP accelerator (such as APC, XCache, or eAccelerator) is almost compulsory for a production server, because it can make PHP run an average 50% faster, with no tradeoff. Make sure you install one of the accelerator extensions to feel the real speed of PHP.

On the other hand, make sure you deactivate any debug utility, such as the Xdebug or APD extension, in your production server.



You might be wondering about the overhead caused by the `mod_rewrite` extension: it is negligible. Of course, loading an image with rewriting rules is slower than loading an image without, but the slowdown is orders of magnitude below the execution of any PHP statement.



When one server is not enough, you can still add another and use load balancing. As long as the `uploads/` directory is shared and you use database storage for sessions, a symfony project will react seamlessly in a load-balanced architecture.

Tweaking the Model

In symfony, the model layer has the reputation of being the slowest part. If benchmarks show that you have to optimize this layer, here are a few possible improvements.

Optimizing Propel or Doctrine Integration

Initializing the model layer (the core ORM classes) takes some time, because of the need to load a few classes and construct various objects. However, because of the way symfony integrates the both ORMs, these initialization tasks occur only when an action actually needs the model—and as late as possible. The ORMs classes will be initialized only when an object of your generated model is autoloaded. This means pages that don't use the model are not penalized by the model layer.

If your entire application doesn't require the use of the model layer, you can also save the initialization of the `sfDatabaseManager` by switching the whole layer off in your `settings.yml`:

```
all:  
  .settings:  
    use_database: false
```

Listing 18-1

Propel enhancements

The generated model classes (in `lib/model/om/`) are already optimized—they don't contain comments, and they benefit from the autoloading system. Relying on autoloading instead of manually including files means that classes are loaded only if it is really necessary. So in case one model class is not needed, having classes autoloaded will save execution time, while the alternative method of using `include` statements won't. As for the comments, they document the use of the generated methods but lengthen the model files—resulting in a minor overhead on slow disks. As the generated method names are pretty explicit, the comments are turned off by default.

These two enhancements are symfony-specific, but you can revert to the Propel defaults by changing two settings in your `propel.ini` file, as follows:

```
propel.builder.addIncludes = true  # Add include statements in generated  
classes  
                                # Instead of relying on the
```

Listing 18-2

```
autoload system
propel.builder.addComments = true # Add comments to generated classes
```

Limits the Number of Objects to Hydrate

When you use a method of a peer class to retrieve objects, your query goes through the hydrating process (creating and populating objects based on the rows of the result of the query). For instance, to retrieve all the rows of the `article` table with Propel, you usually do the following:

Listing 18-3 `$articles = ArticlePeer::doSelect(new Criteria());`

The resulting `$articles` variable is an array of objects of class `Article`. Each object has to be created and initialized, which takes time. This has one major consequence: Contrary to direct database queries, the speed of a Propel query is directly proportional to the number of results it returns. This means your model methods should be optimized to return only a given number of results. When you don't need all the results returned by a `Criteria`, you should limit it with the `setLimit()` and `setOffset()` methods. For instance, if you need only the rows 10 to 20 of a particular query, refine the `Criteria` as in Listing 18-1.

Listing 18-1 - Limiting the Number of Results Returned by a Criteria

Listing 18-4 `$c = new Criteria();
$c->setOffset(10); // Offset of the first record returned
$c->setLimit(10); // Number of records returned
$articles = ArticlePeer::doSelect($c);`

This can be automated by the use of a pager. The `sfPropelPager` object automatically handles the offset and the limit of a Propel query to hydrate only the objects required for a given page.

Minimizing the Number of Queries with Joins

During application development, you should keep an eye on the number of database queries issued by each request. The web debug toolbar shows the number of queries for each page, and clicking the little database icon reveals the SQL code of these queries. If you see the number of queries rising abnormally, it is time to consider using a Join.

Before explaining the Join methods, let's review what happens when you loop over an array of objects and use a Propel getter to retrieve details about a related class, as in Listing 18-2. This example supposes that your schema describes an `article` table with a foreign key to an `author` table.

Listing 18-2 - Retrieving Details About a Related Class in a Loop

Listing 18-5 `// In the action, with Propel
$this->articles = ArticlePeer::doSelect(new Criteria());
// Or with Doctrine
$this->articles = Doctrine::getTable('Article')->findAll();

// Database query issued by doSelect()
SELECT article.id, article.title, article.author_id, ...
FROM article

// In the template

<?php foreach ($articles as $article): ?>
 <?php echo $article->getTitle() ?>,`

```
written by <?php echo $article->getAuthor()->getName() ?></li>
<?php endforeach; ?>
</ul>
```

If the `$articles` array contains ten objects, the `getAuthor()` method will be called ten times, which in turn executes one database query each time it is called to hydrate one object of class `Author`, as in Listing 18-3.

Listing 18-3 - Foreign Key Getters Issue One Database Query

```
// In the template
$article->getAuthor()

// Database query issued by getAuthor()
SELECT author.id, author.name, ...
FROM   author
WHERE  author.id = ?           // ? is article.author_id
```

*Listing
18-6*

So the page of Listing 18-2 will require a total of 11 queries: the one necessary to build the array of `Article` objects, plus the 10 queries to build one `Author` object at a time. This is a lot of queries to display only a list of articles and their author.

How to optimize your queries with Propel

If you were using plain SQL, you would know how to reduce the number of queries to only one by retrieving the columns of the `article` table and those of the `author` table in the same query. That's exactly what the `doSelectJoinAuthor()` method of the `ArticlePeer` class does. It issues a slightly more complex query than a simple `doSelect()` call, but the additional columns in the result set allow Propel to hydrate both `Article` objects and the related `Author` objects. The code of Listing 18-4 displays exactly the same result as Listing 18-2, but it requires only one database query to do so rather than 11 and therefore is faster.

Listing 18-4 - Retrieving Details About Articles and Their Author in the Same Query

```
// In the action
$this->articles = ArticlePeer::doSelectJoinAuthor(new Criteria());

// Database query issued by doSelectJoinAuthor()
SELECT article.id, article.title, article.author_id, ...
      author.id, author.name, ...
FROM   article, author
WHERE  article.author_id = author.id

// In the template (unchanged)
<ul>
<?php foreach ($articles as $article): ?>
  <li><?php echo $article->getTitle() ?>,
    written by <?php echo $article->getAuthor()->getName() ?></li>
<?php endforeach; ?>
</ul>
```

*Listing
18-7*

There is no difference in the result returned by a `doSelect()` call and a `doSelectJoinXXX()` method; they both return the same array of objects (of class `Article` in the example). The difference appears when a foreign key getter is used on these objects afterwards. In the case of `doSelect()`, it issues a query, and one object is hydrated with the result; in the case of `doSelectJoinXXX()`, the foreign object already exists and no query is required, and the process is much faster. So if you know that you will need related objects,

call a `doSelectJoinXXX()` method to reduce the number of database queries—and improve the page performance.

The `doSelectJoinAuthor()` method is automatically generated when you call a `propel-build-model` because of the relationship between the `article` and `author` tables. If there were other foreign keys in the article table structure—for instance, to a category table—the generated `BaseArticlePeer` class would have other Join methods, as shown in Listing 18-5.

Listing 18-5 - Example of Available doSelect Methods for an ArticlePeer Class

```
Listing 18-8 // Retrieve Article objects
doSelect()

// Retrieve Article objects and hydrate related Author objects
doSelectJoinAuthor()

// Retrieve Article objects and hydrate related Category objects
doSelectJoinCategory()

// Retrieve Article objects and hydrate related objects except Author
doSelectJoinAllExceptAuthor()

// Synonym of
doSelectJoinAll()
```

The peer classes also contain Join methods for `doCount()`. The classes with an i18n counterpart (see Chapter 13) provide a `doSelectWithI18n()` method, which behaves the same as Join methods but for i18n objects. To discover the available Join methods in your model classes, you should inspect the generated peer classes in `lib/model/om/`. If you don't find the Join method needed for your query (for instance, there is no automatically generated Join method for many-to-many relationships), you can build it yourself and extend your model.



Of course, a `doSelectJoinXXX()` call is a bit slower than a call to `doSelect()`, so it only improves the overall performance if you use the hydrated objects afterwards.

Optimize your queries with Doctrine

Doctrine comes with its own query language called DQL, for *Doctrine Query Language*. The syntax is very similar to the SQL one, but allows to retrieve objects instead of result set rows. In SQL, you will want to return the columns of the table `article` and `author` in the same query. With the DQL, the solution is quite easy as the only things to do is to add a join statement to the original query, and Doctrine will hydrate your objects instances accordingly. The following code shows how to make a join between the two tables:

```
Listing 18-9 // in the action
Doctrine::getTable('Article')
    ->createQuery('a')
    ->innerJoin('a.Author') // "a.Author" refers to the relation named
    "Author"
    ->execute();

// In the template (unchanged)
<ul>
<?php foreach ($articles as $article): ?>
<li><?php echo $article->getTitle() ?>,
    written by <?php echo $article->getAuthor()->getName() ?></li>
```

```
<?php endforeach; ?>
</ul>
```

Avoid Using Temporary Arrays

When using Propel, objects are already hydrated, so there is no need to prepare a temporary array for the template. Developers not used to ORMs usually fall into this trap. They want to prepare an array of strings or integers, whereas the template can rely directly on an existing array of objects. For instance, imagine that a template displays the list of all the titles of the articles present in the database. A developer who doesn't use OOP would probably write code similar to what is shown in Listing 18-6.

Listing 18-6 - Preparing an Array in the Action Is Useless If You Already Have One

```
// In the action
$articles = ArticlePeer::doSelect(new Criteria());
$titles = array();
foreach ($articles as $article)
{
    $titles[] = $article->getTitle();
}
$this->titles = $titles;

// In the template
<ul>
<?php foreach ($titles as $title): ?>
    <li><?php echo $title ?></li>
<?php endforeach; ?>
</ul>
```

*Listing
18-10*

The problem with this code is that the hydrating is already done by the `doSelect()` call (which takes time), making the `$titles` array superfluous, since you can write the same code as in Listing 18-7. So the time spent to build the `$titles` array could be gained to improve the application performance.

Listing 18-7 - Using an Array of Objects Exempts You from Creating a Temporary Array

```
// In the action
$this->articles = ArticlePeer::doSelect(new Criteria());
// With Doctrine
$this->articles = Doctrine::getTable('Article')->findAll();

// In the template
<ul>
<?php foreach ($articles as $article): ?>
    <li><?php echo $article->getTitle() ?></li>
<?php endforeach; ?>
</ul>
```

*Listing
18-11*

If you feel that you really need to prepare a temporary array because some processing is necessary on objects, the right way to do so is to create a new method in your model class that directly returns this array. For instance, if you need an array of article titles and the number of comments for each article, the action and the template should look like Listing 18-8.

Listing 18-8 - Using a Custom Method to Prepare a Temporary Array

```
// In the action
$this->articles = ArticlePeer::getArticleTitlesWithNbComments();
```

*Listing
18-12*

```
// In the template
<ul>
<?php foreach ($articles as $article): ?>
    <li><?php echo $article['title'] ?> (<?php echo $article['nb_comments'] ?> comments)</li>
<?php endforeach; ?>
</ul>
```

It's up to you to build a fast-processing `getArticleTitlesWithNbComments()` method in the model—for instance, by bypassing the whole object-relational mapping and database abstraction layers.

Bypassing the ORM

When you don't really need objects but only a few columns from various tables, as in the previous example, you can create specific methods in your model that bypass completely the ORM layer. You can directly call the database with PDO, for instance, and return a custom-built array. Listing 18-9 illustrates this idea.

Listing 18-9 - Using Direct PDO Access for Optimized Model Methods, in lib/model/ArticlePeer.php

Listing 18-13

```
// With Propel
class ArticlePeer extends BaseArticlePeer
{
    public static function getArticleTitlesWithNbComments()
    {
        $connection = Propel::getConnection();
        $query = 'SELECT %s as title, COUNT(%s) AS nb_comments FROM %s LEFT
JOIN %s ON %s = %s GROUP BY %s';
        $query = sprintf($query,
            ArticlePeer::TITLE, CommentPeer::ID,
            ArticlePeer::TABLE_NAME, CommentPeer::TABLE_NAME,
            ArticlePeer::ID, CommentPeer::ARTICLE_ID,
            ArticlePeer::ID
        );

        $statement = $connection->prepare($query);
        $statement->execute();

        $results = array();
        while ($resultset = $statement->fetch(PDO::FETCH_OBJ))
        {
            $results[] = array('title' => $resultset->title, 'nb_comments' =>
$resultset->nb_comments);
        }

        return $results;
    }
}

// With Doctrine
class ArticleTable extends Doctrine_Table
{
    public function getArticleTitlesWithNbComments()
    {
        return $this->createQuery('a')
```

```

        ->select('a.title, count(*) as nb_comments')
        ->leftJoin('a.Comments')
        ->groupBy('a.id')
        ->fetchArray();
    }
}

```

When you start building these sorts of methods, you may end up writing one custom method for each action, and lose the benefit of the layer separation—not to mention the fact that you lose database-independence.

Speeding Up the Database

There are many database-specific optimization techniques that can be applied regardless of whether you're using symfony. This section briefly outlines the most common database optimization strategies, but a good knowledge of database engines and administration is required to get the most out of your model layer.



Remember that the web debug toolbar displays the time taken by each query in a page, and that every tweak should be monitored to determine whether it really improves performance.

Table queries are often based on non-primary key columns. To improve the speed of such queries, you should define indexes in your database schema. To add a single column index, add the `index: true` property to the column definition, as in Listing 18-10.

Listing 18-10 - Adding a Single Column Index, in config/schema.yml

```

# Propel schema
propel:
    article:
        id:
        author_id:
        title: { type: varchar(100), index: true }

# Doctrine schema
Article:
    columns:
        author_id: integer
        title: string(100)
    indexes:
        title:
            fields: [title]

```

Listing 18-14

You can use the alternative `index: unique` syntax to define a unique index instead of a classic one. You can also define multiple column indices in `schema.yml` (refer to Chapter 8 for more details about the indexing syntax). You should strongly consider doing this, because it is often a good way to speed up a complex query.

After adding an index to a schema, you should do the same in the database itself, either by issuing an `ADD INDEX` query directly in the database or by calling the `propel-build-all` command (which will not only rebuild the table structure, but also erase all the existing data).



Indexing tends to make `SELECT` queries faster, but `INSERT`, `UPDATE`, and `DELETE` queries are slower. Also, database engines use only one index per query, and they infer the index to

be used for each query based on internal heuristics. Adding an index can sometimes be disappointing in terms of performance boost, so make sure you measure the improvements.

Unless specified otherwise, each request uses a single database connection in symfony, and the connection is closed at the end of the request. You can enable persistent database connections to use a pool of database connections that remain open between queries, by setting `persistent: true` in the `databases.yml` file, as shown in Listing 18-11.

Listing 18-11 - Enabling Persistent Database Connection Support, in config/databases.yml

```
Listing 18-15 prod:
  propel:
    class:           sfPropelDatabase
    param:
      dsn:          mysql:dbname=example;host=localhost
      username:     username
      password:     password
      persistent:   true      # Use persistent connections
```

This may or may not improve the overall database performance, depending on numerous factors. The documentation on the subject is abundant on the Internet. Make sure you benchmark your application performance before and after changing this setting to validate its interest.

MySQL-specific tips

Many settings of the MySQL configuration, found in the `my.cnf` file, may alter database performance. Make sure you read the online documentation⁶⁵ on this subject.

One of the tools provided by MySQL is the slow queries log. All SQL statements that take more than `long_query_time` seconds to execute (this is a setting that can be changed in the `my.cnf`) are logged in a file that is quite difficult to construe by hand, but that the `mysqldumpslow` command summarizes usefully. This is a great tool to detect the queries that require optimizations.

Tweaking the View

According to how you design and implement the view layer, you may notice small slowdowns or speedups. This section describes the alternatives and their tradeoffs.

Using the Fastest Code Fragment

If you don't use the caching system, you have to be aware that an `include_component()` is slightly slower than an `include_partial()`, which itself is slightly slower than a simple PHP `include`. This is because symfony instantiates a view to include a partial and an object of class `sfComponent` to include a component, which collectively add some minor overhead beyond what's required to include the file.

However, this overhead is insignificant, unless you include a lot of partials or components in a template. This may happen in lists or tables, and every time you call an `include_partial()` helper inside a `foreach` statement. When you notice that a large number of partial or component inclusions have a significant impact on your performance, you may consider

⁶⁵ [http://dev.mysql.com/doc/refman/5.0/en\(option-files.html](http://dev.mysql.com/doc/refman/5.0/en(option-files.html)

caching (see Chapter 12), and if caching is not an option, then switch to simple `include` statements.

As for slots, the difference in performance is perceptible. The process time necessary to set and include a slot is negligible—it is equivalent to a variable instantiation. Slots are always cached within the template that includes them.

Speeding Up the Routing Process

As explained in Chapter 9, every call to a link helper in a template asks the routing system to process an internal URI into an external URL. This is done by finding a match between the URI and the patterns of the `routing.yml` file. Symfony does it quite simply: It tries to match the first rule with the given URI, and if it doesn't work, it tries with the following, and so on. As every test involves regular expressions, this is quite time consuming.

There is a simple workaround: Use the rule name instead of the module/action couple. This will tell symfony which rule to use, and the routing system won't lose time trying to match all previous rules.

In concrete terms, consider the following routing rule, defined in your `routing.yml` file:

```
article_by_id:
  url:      /article/:id
  param:    { module: article, action: read }
```

*Listing
18-16*

Then instead of outputting a hyperlink this way:

```
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>
```

*Listing
18-17*

you should use the fastest version:

```
<?php echo link_to('my article', 'article_by_id', array('id' =>
$article->getId())) ?>
```

*Listing
18-18*

The difference starts being noticeable when a page includes a few dozen routed hyperlinks.

Skipping the Template

Usually, a response is composed of a set of headers and content. But some responses don't need content. For instance, some Ajax interactions need only a few pieces of data from the server in order to feed a JavaScript program that will update different parts of the page. For this kind of short response, a set of headers alone is faster to transmit. As discussed in Chapter 11, an action can return only a JSON header. Listing 18-12 reproduces an example from Chapter 11.

Listing 18-12 - Example Action Returning a JSON Header

```
public function executeRefresh()
{
  $output = '{"title": "My basic letter", "name": "Mr Brown"}';
  $this->getResponse()->setHttpHeader("X-JSON", '('. $output .')');

  return sfView::HEADER_ONLY;
}
```

*Listing
18-19*

This skips the template and the layout, and the response can be sent at once. As it contains only headers, it is more lightweight and will take less time to transmit to the user.

Chapter 6 explained another way to skip the template by returning content text directly from the action. This breaks the MVC separation, but it can increase the responsiveness of an action greatly. Check Listing 18-13 for an example.

Listing 18-13 - Example Action Returning Content Text Directly

Listing 18-20

```
public function executeFastAction()
{
    return $this->renderText("<html><body>Hello, World!</body></html>");
}
```

Tweaking the Cache

Chapter 12 already described how to cache parts of a response or all of it. The response cache results in a major performance improvement, and it should be one of your first optimization considerations. If you want to make the most out of the cache system, read further, for this section unveils a few tricks you might not have thought of.

Clearing Selective Parts of the Cache

During application development, you have to clear the cache in various situations:

- When you create a new class: Adding a class to an autoloading directory (one of the project's `lib/` folders) is not enough to have `symfony` find it automatically in non-development environments. You must clear the autoloading configuration cache so that `symfony` browses again all the directories of the `autoload.yml` file and references the location of autoloadable classes—including the new ones.
- When you change the configuration in production: The configuration is parsed only during the first request in production. Further requests use the cached version instead. So a change in the configuration in the production environment (or any environment where `debug` is turned off) doesn't take effect until you clear the cached version of the file.
- When you modify a template in an environment where the template cache is enabled: The valid cached templates are always used instead of existing templates in production, so a template change is ignored until the template cache is cleared or outdated.
- When you update an application with the `project:deploy` command: This case usually covers the three previous modifications.

The problem with clearing the whole cache is that the next request will take quite long to process, because the configuration cache needs to be regenerated. Besides, the templates that were not modified will be cleared from the cache as well, losing the benefit of previous requests.

That means it's a good idea to clear only the cache files that really need to be regenerated. Use the options of the `cache:clear` task to define a subset of cache files to clear, as demonstrated in Listing 18-14.

Listing 18-14 - Clearing Only Selective Parts of the Cache

Listing 18-21

```
// Clear only the cache of the frontend application
$ php symfony cache:clear frontend

// Clear only the HTML cache of the frontend application
$ php symfony cache:clear frontend template
```

```
// Clear only the configuration cache of the frontend application
$ php symfony cache:clear frontend config
```

You can also remove files by hand in the `cache/` directory, or clear template cache files selectively from the action with the `$cacheManager->remove()` method, as described in Chapter 12.

All these techniques will minimize the negative performance impact of any of the changes listed previously.



When you upgrade symfony, the cache is automatically cleared, without manual intervention (if you set the `check_symfony_version` parameter to `true` in `settings.yml`).

Generating Cached Pages

When you deploy a new application to production, the template cache is empty. You must wait for users to visit a page once for this page to be put in the cache. In critical deployments, the overhead of page processing is not acceptable, and the benefits of caching must be available as soon as the first request is issued.

The solution is to automatically browse the pages of your application in the staging environment (where the configuration is similar to the one in production) to have the template cache generated, then to transfer the application with the cache to production.

To browse the pages automatically, one option is to create a shell script that looks through a list of external URLs with a browser (`curl` for instance). But there is a better and faster solution: a PHP script using the `sfBrowser` object, already discussed in Chapter 15. That's an internal browser written in PHP (and used by `sfTestFunctional` for functional tests). It takes an external URL and returns a response, but the interesting thing is that it triggers the template cache just like a regular browser. As it only initializes symfony once and doesn't pass by the HTTP transport layer, this method is a lot faster.

Listing 18-15 shows an example script used to generate template cache files in a staging environment. Launch it by calling `php generate_cache.php`.

Listing 18-15 - Generating the Template Cache, in `generate_cache.php`

```
require_once(dirname(__FILE__).'/../config/
ProjectConfiguration.class.php');
$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'staging',
false);
sfContext::createInstance($configuration);

// Array of URLs to browse
$uris = array(
  '/foo/index',
  '/foo/bar/id/1',
  '/foo/bar/id/2',
  ...
);

$b = new sfBrowser();
foreach ($uris as $uri)
{
  $b->get($uri);
}
```

*Listing
18-22*

Using a Database Storage System for Caching

The default storage system for the template cache in symfony is the file system: Fragments of HTML or serialized response objects are stored under the `cache/` directory of a project. Symfony proposes an alternative way to store cache: a SQLite database. Such a database is a simple file that PHP natively knows how to query very efficiently.

To tell symfony to use SQLite storage instead of file system storage for the template cache, open the `factories.yml` file and edit the `view_cache` entry as follows:

Listing 18-23

```
view_cache:
    class: sfSQLiteCache
    param:
        database: %SF_TEMPLATE_CACHE_DIR%/cache.db
```

The benefits of using SQLite storage for the template cache are faster read and write operations when the number of cache elements is important. If your application makes heavy use of caching, the template cache files end up scattered in a deep file structure; in this case, switching to SQLite storage will increase performance. In addition, clearing the cache on file system storage may require a lot of files to be removed from the disk; this operation may last a few seconds, during which your application is unavailable. With a SQLite storage system, the cache clearing process results in a single file operation: the deletion of the SQLite database file. Whatever the number of cache elements currently stored, the operation is instantaneous.

Bypassing Symfony

Perhaps the best way to speed symfony up is to bypass it completely... this is said only partly in jest. Some pages don't change and don't need to be reprocessed by the framework at each request. The template cache is already here to speed up the delivery of such pages, but it still relies on symfony.

A couple of tricks described in Chapter 12 allow you to bypass symfony completely for some pages. The first one involves the use of HTTP 1.1 headers for asking the proxies and client browsers to cache the page themselves, so that they don't request it again the next time the page is needed. The second one is the super fast cache (automated by the `sfSuperCachePlugin` plug-in), which consists of storing a copy of the response in the `web/` directory and modifying the rewriting rules so that Apache first looks for a cached version before handing a request to symfony.

Both these methods are very effective, and even if they only apply to static pages, they will take the burden of handling these pages off from symfony, and the server will then be fully available to deal with complex requests.

Caching the Result of a Function Call

If a function doesn't rely on context-sensitive values nor on randomness, calling it twice with the same parameters should return the same result. That means the second call could very well be avoided if the result had been stored the first time. That's exactly what the `sfFunctionCache` class does. This class has a `call()` method, which expects a callable and an array of parameters as its arguments. When called, this method creates an md5 hash with all its arguments and looks in the cache for a key named by this hash. If such a key is found, the function returns the result stored in the cache. If not, the `sfFunctionCache` executes the function, stores the result in the cache, and returns it. So the second execution of Listing 18-16 will be faster than the first one.

Listing 18-16 - Caching the Result of a Function

```
$cache = new sfFileCache(array('cache_dir' =>
sfConfig::get('sf_cache_dir').'/function'));
$fc = new sfFunctionCache($cache);
$result1 = $fc->call('cos', array(M_PI));
$result2 = $fc->call('preg_replace', array('/\s\s+/', ' ', $input));
```

Listing
18-24

The `sfFunctionCache` constructor expects a cache object. The first argument of the `call()` method must be a callable, so it can be a function name, an array of a class name and static method name, or an array of an object name and public method name. As for the other argument of the `call()` method, it's an array of arguments that will be passed to the callable.



If you use a file based cache object as in the example, it's better to give a cache directory under the `cache/` directory, as it will be cleanup automatically by the `cache:clear` task. If you store the function cache somewhere else, it will not be cleared automatically when you clear the cache through the command line.

Caching Data in the Server

PHP accelerators provide special functions to store data in memory so that you can reuse it across requests. The problem is that they all have a different syntax, and each has its own specific way of performing this task. The symfony cache classes abstract all these differences and works with whatever accelerator you are using. See its syntax in Listing 18-17.

Listing 18-17 - Using a PHP accelerator to cache data

```
$cache = new sfAPCCache();

// Storing data in the cache
$cache->set($name, $value, $lifetime);

// Retrieving data
$value = $cache->get($name);

// Checking if a piece of data exists in the cache
$value_exists = $cache->has($name);

// Clear the cache
$cache->clear();
```

Listing
18-25

The `set()` method returns `false` if the caching didn't work. The cached value can be anything (a string, an array, an object); the `sfAPCCache` class will deal with the serialization. The `get()` method returns `null` if the required variable doesn't exist in the cache.



If you want to go further into memory caching, make sure you take a look at the `sfMemcacheCache` class. It provides the same interface as the other cache classes and it can help decrease the database load on load-balanced applications.

Deactivating the Unused Features

The default symfony configuration activates the most common features of a web application. However, if you happen to not need all of them, you should deactivate them to save the time their initialization takes on each request.

For instance, if your application doesn't use the session mechanism, or if you want to start the session handling by hand, you should turn the `auto_start` setting to `false` in the `storage` key of the `factories.yml` file, as in Listing 18-18.

Listing 18-18 - Turning Sessions Off, in frontend/config/factories.yml

```
Listing 18-26 all:
  storage:
    class: sfSessionStorage
    param:
      auto_start: false
```

The same applies for the database feature (as explained in the “Tweaking the Model” section earlier in this chapter). If your application makes no use of a database, deactivate it for a small performance gain, this time in the `settings.yml` file (see Listing 18-19).

Listing 18-19 - Turning Database Features Off, in frontend/config/settings.yml

```
Listing 18-27 all:
  .settings:
    use_database:      false      # Database and model features
```

As for the security features (see Chapter 6), you can deactivate them in the `filters.yml` file, as shown in Listing 18-20.

Listing 18-20 - Turning Features Off, in frontend/config/filters.yml

```
Listing 18-28 rendering: ~
security:
  enabled: false

# generally, you will want to insert your own filters here

cache:   ~
execution: ~
```

Some features are useful only in development, so you should not activate them in production. This is already the case by default, since the production environment in symfony is really optimized for performance. Among the performance-impacting development features, the debug mode is the most severe. As for the symfony logs, the feature is also turned off in production by default.

You may wonder how to get information about failed requests in production if logging is disabled, and argue that problems arise not only in development. Fortunately, symfony can use the `sfErrorLoggerPlugin` plug-in, which runs in the background in production and logs the details of 404 and 500 errors in a database. It is much faster than the file logging feature, because the plug-in methods are called only when a request fails, while the logging mechanism, once turned on, adds a non-negligible overhead whatever the level. Check the installation instructions and manual⁶⁶.



Make sure you regularly check the server error logs—they also contain very valuable information about 404 and 500 errors.

66. <http://www.symfony-project.org/plugins/sfErrorLoggerPlugin>

Optimizing Your Code

It's also possible to speed up your application by optimizing the code itself. This section offers some insight regarding how to do that.

Core Compilation

Loading ten files requires more I/O operations than loading one long file, especially on slow disks. Loading a very long file requires more resources than loading a smaller file—especially if a large share of the file content is of no use for the PHP parser, which is the case for comments.

So merging a large number of files and stripping out the comments they contain is an operation that improves performance. Symfony already does that optimization; it's called the core compilation. At the beginning of the first request (or after the cache is cleared), a symfony application concatenates all the core framework classes (`sfActions`, `sfRequest`, `sfView`, and so on) into one file, optimizes the file size by removing comments and double blanks, and saves it in the cache, in a file called `config_core_compile.yml.php`. Each subsequent request only loads this single optimized file instead of the 30 files that compose it.

If your application has classes that must always be loaded, and especially if they are big classes with lots of comments, it may be beneficial to add them to the core compile file. To do so, just add a `core_compile.yml` file in your application `config/` directory, and list in it the classes that you want to add, as in Listing 18-21.

Listing 18-21 - Adding Your Classes to the Core Compile File, in frontend/config/core_compile.yml

```
- %SF_ROOT_DIR%/lib/myClass.class.php
- %SF_ROOT_DIR%/apps/frontend/lib/myToolkit.class.php
- %SF_ROOT_DIR%/plugins/myPlugin/lib/myPluginCore.class.php
...
```

Listing 18-29

The `project:optimize` Task

Symfony also offers another optimization tool, the `project:optimize` task. It applies various optimization strategies to the symfony and application code, which may further speed up the execution.

```
$ php symfony project:optimize frontend prod
```

Listing 18-30

If you want to see the optimization strategies implemented in the task, have a look at task source code.

Summary

Symfony is already a very optimized framework and is able to handle high-traffic websites without a problem. But if you really need to optimize your application's performance, tweaking the configuration (whether the server configuration, the PHP configuration, or the application settings) will gain you a small boost. You should also follow good practices to write efficient model methods; and since the database is often a bottleneck in web applications, this point should require all your attention. Templates can also benefit from a few tricks, but the best boost will always come from caching. Finally, don't hesitate to look at existing plug-ins, since some of them provide innovative techniques to further speed up the delivery of web pages (`sfSuperCache`, `project:optimize`).

Chapter 19

Mastering Symfony's Configuration Files

Now that you know symfony very well, you are already able to dig into its code to understand its core design and discover new hidden abilities. But before extending the symfony classes to match your own requirements, you should take a closer look at some of the configuration files. Many features are already built into symfony and can be activated by just changing configuration settings. This means that you can tweak the symfony core behavior without overriding its classes. This chapter takes you deep into the configuration files and their powerful capabilities.

Symfony Settings

The `frontend/config/settings.yml` file contains the main symfony configuration for the frontend application. You have already seen the function of many settings from this file in the previous chapters, but let's revisit them.

As explained in Chapter 5, this file is environment-dependent, which means that each setting can take a different value for each environment. Remember that each parameter defined in this file is accessible from inside the PHP code via the `sfConfig` class. The parameter name is the setting name prefixed with `sf_`. For instance, if you want to get the value of the `cache` parameter, you just need to call `sfConfig::get('sf_cache')`.

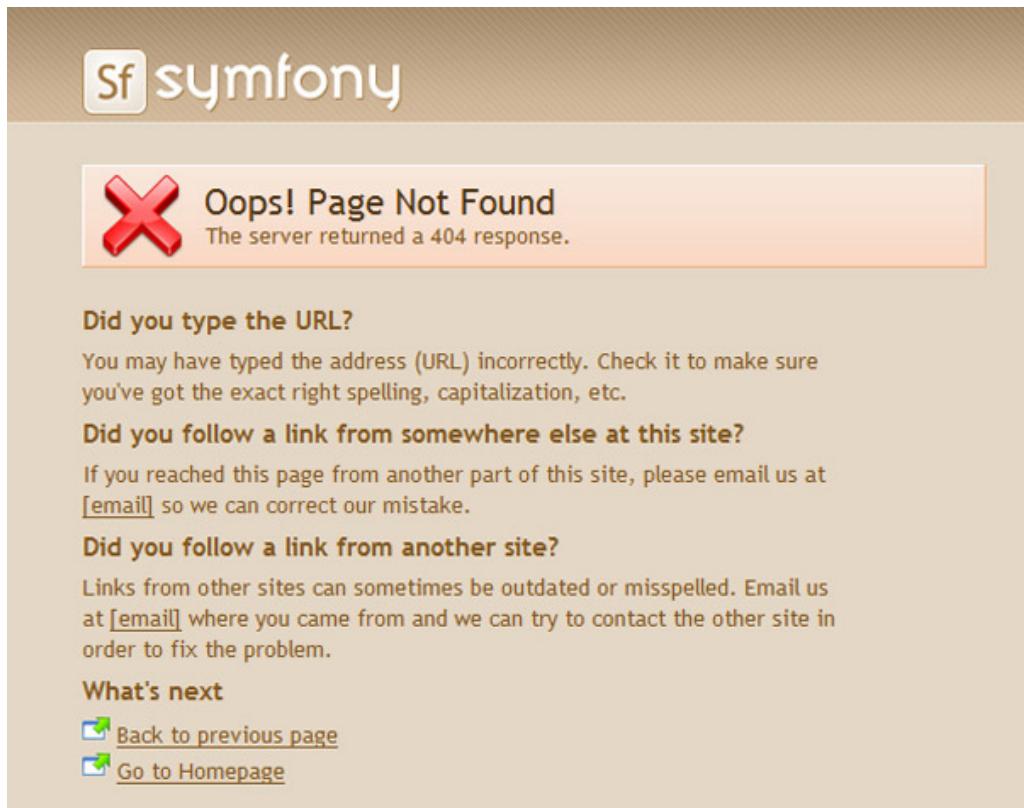
Default Modules and Actions

Symfony provides default pages for special situations. In the case of a routing error, symfony executes an action of the `default` module, which is stored in the `sfConfig::get('sf_symfony_lib_dir')/controller/default/` directory. The `settings.yml` file defines which action is executed depending on the error:

- `error_404_module` and `error_404_action`: Action called when the URL entered by the user doesn't match any route or when an `sfError404Exception` occurs. The default value is `default/error404`.
- `login_module` and `login_action`: Action called when a nonauthenticated user tries to access a page defined as `secure` in `security.yml` (see Chapter 6 for details). The default value is `default/login`.
- `secure_module` and `secure_action`: Action called when a user doesn't have the credentials required for an action. The default value is `default/secure`.
- `module_disabled_module` and `module_disabled_action`: Action called when a user requests a module declared as disabled in `module.yml`. The default value is `default/disabled`.

Before deploying an application to production, you should customize these actions, because the `default` module templates include the symfony logo on the page. See Figure 19-1 for a screenshot of one of these pages, the error 404 page.

Figure 19-1 - Default 404 error page



You can override the default pages in two ways:

- You can create your own default module in the application's `modules/` directory, override all the actions defined in the `settings.yml` file (`index`, `error404`, `login`, `secure`, `disabled`) and all the related templates (`indexSuccess.php`, `error404Success.php`, `loginSuccess.php`, `secureSuccess.php`, `disabledSuccess.php`).
- You can change the default module and action settings of the `settings.yml` file to use pages of your application.

Two other pages bear a symfony look and feel, and they also need to be customized before deployment to production. These pages are not in the `default` module, because they are called when symfony cannot run properly. Instead, you will find these default pages in the `sfConfig::get('sf_symfony_lib_dir')/exception/data/` directory:

- `error.html.php`: Page called when an internal server error occurs in the production environment. In other environments (where `debug` is set to `true`), when an error occurs, symfony displays the full execution stack and an explicit error message (see Chapter 16 for details).
- `unavailable.php`: Page called when a user requests a page while the application is disabled (with the `project:disable` task). It is also called while the cache is being cleared (that is, between a call to the `cache:clear` task and the end of this task execution). On systems with a very large cache, the cache-clearing process can take several seconds. Symfony cannot execute a request with a partially cleared cache, so requests received before the end of the process are redirected to this page.

To customize these pages, simply create `error/error.html.php` and `unavailable.php` in your project or application's `config/` directory. Symfony will use these templates instead of its own.



To have requests redirected to the `unavailable.php` page when needed, you need to set the `check_lock` setting to `true` in the application `settings.yml`. The check is deactivated by default, because it adds a very slight overhead for every request.

Optional Feature Activation

Some parameters of the `settings.yml` file control optional framework features that can be enabled or disabled. Deactivating unused features boosts performances a bit, so make sure to review the settings listed in Table 19-1 before deploying your application.

Table 19-1 - Optional Features Set Through `settings.yml`

Parameter	Description	Default Value
<code>use_database</code>	Enables the database manager. Set it to <code>false</code> if <code>true</code> you don't use a database.	
<code>i18n</code>	Enables interface translation (see Chapter 13). Set it to <code>true</code> for multilingual applications.	<code>false</code>
<code>logging_enabled</code>	Enables logging of symfony events. Set it to <code>false</code> when you want to turn symfony logging off completely.	<code>true</code>
<code>escaping_strategy</code>	Enables the output escaping feature (see Chapter 7). Set it to <code>true</code> if you want data passed to your templates to be escaped.	<code>true</code>
<code>cache</code>	Enables template caching (see Chapter 12). Set it to <code>true</code> if one of your modules includes <code>cache.yml</code> file. The cache filter (<code>sfCacheFilter</code>) is enabled only if it is on.	<code>false</code> in development, <code>true</code> in production
<code>web_debug</code>	Enables the web debug toolbar for easy debugging (see Chapter 16). Set it to <code>true</code> to display the toolbar on every page.	<code>true</code> in development, <code>false</code> in production
<code>check_symfony_version</code>	Enables the check of the symfony version for every request. Set it to <code>true</code> for automatic cache clearing after a framework upgrade. Leave it set to <code>false</code> if you always clear the cache after an upgrade.	<code>false</code>
<code>check_lock</code>	Enables the application lock system, triggered by the <code>cache:clear</code> and <code>project:disable</code> tasks (see the previous section). Set it to <code>true</code> to have all requests to disabled applications redirected to the <code>sfConfig::get('sf_symfony_lib_dir')/exception/data/unavailable.php</code> page.	<code>false</code>
<code>compressed</code>	Enables PHP response compression. Set it to <code>true</code> to compress the outgoing HTML via the PHP compression handler.	<code>false</code>

Feature Configuration

Symfony uses some parameters of `settings.yml` to alter the behavior of built-in features such as form validation, cache, and third-party modules.

Output Escaping Settings

Output escaping settings control the way the variables are accessible in the template (see Chapter 7). The `settings.yml` file includes two settings for this feature:

- The `escaping_strategy` setting can take the value `true`, or `false`.
- The `escaping_method` setting can be set to `ESC_RAW`, `ESC_SPECIALCHARS`, `ESC_ENTITIES`, `ESC_JS`, or `ESC_JS_NO_ENTITIES`.

Routing Settings

The routing settings (see Chapter 9) are defined in `factories.yml`, under the `routing` key. Listing 19-1 show the default routing configuration.

Listing 19-1 - Routing Configuration Settings, in frontend/config/factories.yml

```
routing:
    class: sfPatternRouting
    param:
        load_configuration: true
        suffix: .
        default_module: default
        default_action: index
        variable_prefixes: [':']
        segment_separators: [ '/', '. ' ]
        variable_regex: '[\w\d]+'
        debug: %SF_DEBUG%
        logging: %SF_LOGGING_ENABLED%
        cache:
            class: sfFileCache
            param:
                automatic_cleaning_factor: 0
                cache_dir: %SF_CONFIG_CACHE_DIR%/routing
                lifetime: 31556926
                prefix: %SF_APP_DIR%
```

*Listing
19-1*

- The `suffix` parameter sets the default suffix for generated URLs. The default value is a period (.), and it corresponds to no suffix. Set it to `.html`, for instance, to have all generated URLs look like static pages.
- When a routing rule doesn't define the `module` or the `action` parameter, values from the `factories.yml` are used instead:
 - `default_module`: Default module request parameter. Defaults to the `default` module.
 - `default_action`: Default action request parameter. Defaults to the `index` action.
- By default, route patterns identify named wildcards by a colon (:) prefix. But if you want to write your rules in a more PHP-friendly syntax, you can add the dollar (\$) sign in the `variable_prefixes` array. That way, you can write a pattern like '`/article/$year/$month/$day/$title`' instead of '`/article/:year/:month/:day/:title`'.
- The pattern routing will identify named wildcards between separators. The default separators are the slash and the dot, but you can add more if you want in the

`segment_separators` parameter. For instance, if you add the dash (-), you can write a pattern like '/article/:year-:month-:day/:title'.

- The pattern routing uses its own cache, in production mode, to speed up conversions between external URLs and internal URIs. By default, this cache uses the filesystem, but you can use any cache class, provided that you declare the class and its settings in the `cache` parameter. See Chapter 15 for the list of available cache storage classes. To deactivate the routing cache in production, set the `debug` parameter to `true`.

These are only the settings for the `sfPatternRouting` class. You can use another class for your application routing, either your own or one of symfony's routing factories (`sfNoRouting` and `sfPathInfoRouting`). With either of these two factories, all external URLs look like 'module/action?key1=param1'. No customization possible—but it's fast. The difference is that the first uses PHP's GET, and the second uses PATH_INFO. Use them mainly for backend interfaces.

There is one additional parameter related to routing, but this one is stored in `settings.yml`:

- `no_script_name` enables the front controller name in generated URLs. The `no_script_name` setting can be on only for a single application in a project, unless you store the front controllers in various directories and alter the default URL rewriting rules. It is usually on for the production environment of your main application and off for the others.

Form Validation Settings



The features described in this section are deprecated since symfony 1.1 and only work if you enable the `sfCompat10` plugin.

Form validation settings control the way error messages output by the `Validation` helpers look (see Chapter 10). These errors are included in `<div>` tags, and they use the `validation_error_class` setting as a `class` attribute and the `validation_error_id_prefix` setting to build up the `id` attribute. The default values are `form_error` and `error_for_`, so the attributes output by a call to the `form_error()` helper for an input named `foobar` will be `class="form_error"` `id="error_for_foobar"`.

Two settings determine which characters precede and follow each error message: `validation_error_prefix` and `validation_error_suffix`. You can change them to customize all error messages at once.

Cache Settings

Cache settings are defined in `cache.yml` for the most part, except for two in `settings.yml`: `cache` enables the template cache mechanism, and `etag` enables ETag handling on the server side (see Chapter 15). You can also specify which storage to use for two all cache systems (the view cache, the routing cache, and the i18n cache) in `factories.yml`. Listing 19-2 show the default view cache factory configuration.

Listing 19-2 - View Cache Configuration Settings, in `frontend/config/factories.yml`

```
Listing 19-2
view_cache:
  class: sfFileCache
  param:
    automatic_cleaning_factor: 0
    cache_dir: %SF_TEMPLATE_CACHE_DIR%
    lifetime: 86400
    prefix: %SF_APP_DIR%/template
```

The `class` can be any of `sfFileCache`, `sfAPCCache`, `sfEAcceleratorCache`, `sfxCacheCache`, `sfMemcacheCache`, and `sfSQLiteCache`. It can also be your own custom class, provided it extends `sfCache` and provides the same generic methods for setting, retrieving and deleting a key in the cache. The factory parameters depend on the class you choose, but there are constants:

- `lifetime` defines the number of seconds after which a cache part is removed
- `prefix` is a prefix added to every cache key (use the environment in the prefix to use different cache depending on the environment). Use the same prefix for two applications if you want their cache to be shared.

Then, for each particular factory, you have to define the location of the cache storage.

- for `sfFileCache`, the `cache_dir` parameter locates the absolute path to the cache directory
- `sfAPCCache`, `sfEAcceleratorCache`, and `sfxCacheCache` don't take any location parameter, since they use PHP native functions for communicating with APC, EAccelerator or the XCache cache systems
- for `sfMemcacheCache`, enter the hostname of the Memcached server in the `host` parameter, or an array of hosts in the `servers` parameter
- for `sfSQLiteCache`, the absolute path to the SQLite database file should be entered in the `database` parameter

For additional parameters, check the API documentation of each cache class.

The view is not the only component to be able to use a cache. Both the `routing` and the `I18N` factories offer a `cache` parameter in which you can set any cache factory, just like the view cache. For instance, Listing 19-1 shows of the routing uses the file cache for its speedup tactics by default, but you can change it to whatever you want.

Logging Settings

Two logging settings (see Chapter 16) are stored in `settings.yml`:

- `error_reporting` specifies which events are logged in the PHP logs. By default, it is set to `E_PARSE | E_COMPILE_ERROR | E_ERROR | E_CORE_ERROR | E_USER_ERROR` for the production environment (so the logged events are `E_PARSE`, `E_COMPILE_ERROR`, `E_ERROR`, `E_CORE_ERROR`, and `E_USER_ERROR`) and to `E_ALL | E_STRICT` for the development environment.
- The `web_debug` setting activates the web debug toolbar. Set it to `true` only in the development and test environments.

Paths to Assets

The `settings.yml` file also stores paths to assets. If you want to use another version of the asset than the one bundled with symfony, you can change these path settings:

- Files needed by the administration generator stored in `admin_web_dir`
- Files needed by the web debug toolbar stored in `web_debug_web_dir`

Default Helpers

Default helpers, loaded for every template, are declared in the `standard_helpers` setting (see Chapter 7). By default, these are the `Partial` and `Cache` helper groups. If you use a helper group in all templates of an application, adding its name to the `standard_helpers` setting saves you the hassle of declaring it with `use_helper()` on each template.

Activated Modules

Activated modules from plug-ins or from the symfony core are declared in the `enabled_modules` parameter. Even if a plug-in bundles a module, users can't request this module unless it is declared in `enabled_modules`. The `default` module, which provides the default symfony pages (congratulations, page not found, and so on), is the only enabled module by default.

Character Set

The character set of the responses is a general setting of the application, because it is used by many components of the framework (templates, output escaper, helpers, and so on). Defined in the `charset` setting, its default (and advised) value is `utf-8`.

Adding Your application settings

The `settings.yml` file defines symfony settings for an application. As discussed in Chapter 5, when you want to add new parameters, the best place to do so is in the `frontend/config/app.yml` file. This file is also environment-dependent, and the settings it defines are available through the `sfConfig` class with the `app_` prefix.

Listing 19-3

```
all:
  creditcards:
    fake:          false  # app_creditcards_fake
    visa:          true   # app_creditcards_visa
    americanexpress: true  # app_creditcards_amERICANEXPRESS
```

You can also write an `app.yml` file in the project configuration directory, and this provides a way to define custom project settings. The configuration cascade also applies to this file, so the settings defined in the application `app.yml` file override the ones defined at the project level.

Extending the Autoloading Feature

The autoloading feature, briefly explained in Chapter 2, exempts you from requiring classes in your code if they are located in specific directories. This means that you can just let the framework do the job for you, allowing it to load only the necessary classes at the appropriate time, and only when needed.

The `autoload.yml` file lists the paths in which autoloaded classes are stored. The first time this configuration file is processed, symfony parses all the directories referenced in the file. Each time a file ending with `.php` is found in one of these directories, the file path and the class names found in this file are added to an internal list of autoloading classes. This list is saved in the cache, in a file called `config/config_autoload.yml.php`. Then, at runtime, when a class is used, symfony looks in this list for the class path and includes the `.php` file automatically.

Autoloading works for all `.php` files containing classes and/or interfaces.

By default, classes stored in the following directories in your projects benefit from the autoloading automatically:

- `myproject/lib/`
- `myproject/lib/model`
- `myproject/apps/frontend/lib/`
- `myproject/apps/frontend/modules/mymodule/lib`

There is no `autoload.yml` file in the default application configuration directory. If you want to modify the framework settings—for instance, to autoload classes stored somewhere else in your file structure—create an empty `autoload.yml` file and override the settings of `sfConfig::get('sf_symfony_lib_dir')/config/config/autoload.yml` or add your own.

The `autoload.yml` file must start with an `autoload:` key and list the locations where symfony should look for classes. Each location requires a label; this gives you the ability to override symfony's entries. For each location, provide a name (it will appear as a comment in `config/autoload.yml.php`) and an absolute path. Then define if the search must be recursive, which directs symfony to look in all the subdirectories for `.php` files, and exclude the subdirectories you want. Listing 19-3 shows the locations used by default and the file syntax.

Listing 19-3 - Default Autoloading Configuration, in `sfConfig::get('sf_symfony_lib_dir')/config/config/autoload.yml`

```
autoload:
  # plugins
  plugins_lib:
    name:           plugins lib
    path:          %SF_PLUGINS_DIR%/*/lib
    recursive:     true

  plugins_module_lib:
    name:           plugins module lib
    path:          %SF_PLUGINS_DIR%/*/modules/*/*/lib
    prefix:        2
    recursive:     true

  # project
  project:
    name:           project
    path:          %SF_LIB_DIR%
    recursive:     true
    exclude:       [model, symfony]

  project_model:
    name:           project model
    path:          %SF_LIB_DIR%/model
    recursive:     true

  # application
  application:
    name:           application
    path:          %SF_APP_LIB_DIR%
    recursive:     true

  modules:
    name:           module
    path:          %SF_APP_DIR%/modules/*/*/lib
    prefix:        1
    recursive:     true
```

Listing 19-4

A rule path can contain wildcards and use the file path parameters defined in the configuration classes (see the next section). If you use these parameters in the configuration file, they must appear in uppercase and begin and end with %.

Editing your own `autoload.yml` will add new locations to symfony's autoloading, but you may want to extend this mechanism and add your own autoloading handler to symfony's handler. As symfony uses the standard `spl_autoload_register()` function to manage class autoloading, you can register more callbacks in the application configuration class:

Listing 19-5

```
class frontendConfiguration extends sfApplicationConfiguration
{
    public function initialize()
    {
        parent::initialize(); // load symfony autoloading first

        // insert your own autoloading callables here
        spl_autoload_register(array('myToolkit', 'autoload'));
    }
}
```

When the PHP autoloading system encounters a new class, it will first try the symfony autoloading method (and use the locations defined in `autoload.yml`). If it doesn't find a class definition, all the other callables registered with `spl_autoload_register()` will be called, until the class is found. So you can add as many autoloading mechanisms as you want—for instance, to provide a bridge to other framework components (see Chapter 17).

Custom File Structure

Each time the framework uses a path to look for something (from core classes to templates, plug-ins, configurations, and so on), it uses a path variable instead of an actual path. By changing these variables, you can completely alter the directory structure of a symfony project, and adapt to the file organization requirements of any client.



Customizing the directory structure of a symfony project is possible but not necessarily a good idea. One of the strengths of a framework like symfony is that any web developer can look at a project built with it and feel at home, because of the respect for conventions. Make sure you consider this issue before deciding to use your own directory structure.

The Basic File Structure

The path variables are defined in the `sfProjectConfiguration` and `sfApplicationConfiguration` classes and stored in the `sfConfig` object. Listing 19-4 shows a listing of the path variables and the directory they reference.

Listing 19-4 - Default File Structure Variables, defined in `sfProjectConfiguration` and `sfApplicationConfiguration`

Listing 19-6

```
sf_root_dir          # myproject/
sf_apps_dir          #   apps/
sf_app_dir           #     frontend/
sf_app_config_dir   #       config/
sf_app_i18n_dir     #       i18n/
sf_app_lib_dir       #       lib/
sf_app_module_dir   #       modules/
sf_app_template_dir #       templates/
sf_cache_dir         #       cache/
sf_app_base_cache_dir #       frontend/
sf_app_cache_dir    #       prod/
sf_template_cache_dir #       templates/
```

```

sf_i18n_cache_dir      #      i18n/
sf_config_cache_dir    #      config/
sf_test_cache_dir      #      test/
sf_module_cache_dir    #      modules/
sf_config_dir          #      config/
sf_data_dir            #      data/
sf_lib_dir              #      lib/
sf_log_dir              #      log/
sf_test_dir             #      test/
sf_plugins_dir         #      plugins/
sf_web_dir              #      web/
sf_upload_dir           #      uploads/

```

Every path to a key directory is determined by a parameter ending with `_dir`. Always use the path variables instead of real (relative or absolute) file paths, so that you will be able to change them later, if necessary. For instance, when you want to move a file to the `uploads/` directory in an application, you should use `sfConfig::get('sf_upload_dir')` for the path instead of `sfConfig::get('sf_root_dir').'/web/uploads/'`.

Customizing the File Structure

You will probably need to modify the default project file structure 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. By overriding the `sf_XXX_dir` variables with `sfConfig`, you can make symfony work for a totally different directory structure than the default structure. The best place to do this is in the application `ProjectConfiguration` class for project directories, or `XXXConfiguration` class for applications directories.

For instance, if you want all applications to share a common directory for the template layouts, add this line to the `setup()` method of the `ProjectConfiguration` class to override the `sf_app_template_dir` settings:

```
sfConfig::set('sf_app_template_dir',
sfConfig::get('sf_root_dir').DIRECTORY_SEPARATOR.'templates');
```

Listing 19-7



Even if you can change your project directory structure by calling `sfConfig::set()`, it's better to use the dedicated methods defined by the project and application configuration classes if possible as they take care of changing all the related paths. For example, the `setCacheDir()` method changes the following constants: `sf_cache_dir`, `sf_app_base_cache_dir`, `sf_app_cache_dir`, `sf_template_cache_dir`, `sf_i18n_cache_dir`, `sf_config_cache_dir`, `sf_test_cache_dir`, and `sf_module_cache_dir`.

Modifying the Project Web Root

All the paths built in the configuration classes rely on the project root directory, which is determined by the `ProjectConfiguration` file included in the front controller. Usually, the root directory is one level above the `web/` directory, but you can use a different structure. Suppose that your main directory structure is made of two directories, one public and one private, as shown in Listing 19-5. This typically happens when hosting a project on a shared host.

Listing 19-5 - Example of Custom Directory Structure for a Shared Host

```
symfony/      # Private area
  apps/
```

Listing 19-8

```
config/
...
www/      # Public area
images/
css/
js/
index.php
```

In this case, the root directory is the `symfony/` directory. So the `index.php` front controller simply needs to include the `config/ProjectConfiguration.class.php` file as follows for the application to work:

Listing 19-9 require_once(dirname(__FILE__).'/../symfony/config/ProjectConfiguration.class.php');

In addition, use the `setWebDir()` method to change the public area from the usual `web/` to `www/`, as follows:

Listing 19-10

```
class ProjectConfiguration extends sfProjectConfiguration
{
    public function setup()
    {
        // ...

        $this->setWebDir($this->getRootDir().'/../www');
    }
}
```

Understanding Configuration Handlers

Each configuration file has a handler. The job of configuration handlers is to manage the configuration cascade, and to do the translation between the configuration files and the optimized PHP code executable at runtime.

Default Configuration Handlers

The default handler configuration is stored in `sfConfig::get('sf_symfony_lib_dir')/config/config_handlers.yml`. This file links the handlers to the configuration files according to a file path. Listing 19-6 shows an extract of this file.

Listing 19-6 - Extract of sfConfig::get('sf_symfony_lib_dir')/config/config_handlers.yml

Listing 19-11

```
config/settings.yml:
    class:    sfDefineEnvironmentConfigHandler
    param:
        prefix: sf_

config/app.yml:
    class:    sfDefineEnvironmentConfigHandler
    param:
        prefix: app_

config/filters.yml:
    class:    sfFilterConfigHandler
```

```
modules/*config/module.yml:
  class:    sfDefineEnvironmentConfigHandler
  param:
    prefix: mod_
    module: yes
```

For each configuration file (`config_handlers.yml` identifies each file by a file path with wildcards), the handler class is specified under the `class` key.

The settings of configuration files handled by `sfDefineEnvironmentConfigHandler` can be made available directly in the code via the `sfConfig` class, and the `param` key contains a `prefix` value.

You can add or modify the handlers used to process each configuration file—for instance, to use INI or XML files instead of YAML files.



The configuration handler for the `config_handlers.yml` file is `sfRootConfigHandler` and, obviously, it cannot be changed.

If you ever need to modify the way the configuration is parsed, create an empty `config_handlers.yml` file in your application's `config/` folder and override the `class` lines with the classes you wrote.

Adding Your Own Handler

Using a handler to deal with a configuration file provides two important benefits:

- The configuration file is transformed into executable PHP code, and this code is stored in the cache. This means that the configuration is parsed only once in production, and the performance is optimal.
- The configuration file can be defined at different levels (project and application) and the final parameter values will result from a cascade. So you can define parameters at a project level and override them on a per-application basis.

If you feel like writing your own configuration handler, follow the example of the structure used by the framework in the `sfConfig::get('sf_symfony_lib_dir')/config/` directory.

Let's suppose that your application contains a `myMapAPI` class, which provides an interface to a third-party web service delivering maps. This class needs to be initialized with a URL and a user name, as shown in Listing 19-7.

Listing 19-7 - Example of Initialization of the myMapAPI Class

```
$mapApi = new myMapAPI();
$mapApi->setUrl($url);
$mapApi->setUser($user);
```

*Listing
19-12*

You may want to store these two parameters in a custom configuration file called `map.yml`, located in the application `config/` directory. This configuration file might contain the following:

```
api:
  url: map.api.example.com
  user: foobar
```

*Listing
19-13*

In order to transform these settings into code equivalent to Listing 19-7, you must build a configuration handler. Each configuration handler must extend `sfConfigHandler` and provide an `execute()` method, which expects an array of file paths to configuration files as a

parameter, and must return data to be written in a cache file. Handlers for YAML files should extend the `sfYamlConfigHandler` class, which provides additional facilities for YAML parsing. For the `map.yml` file, a typical configuration handler could be written as shown in Listing 19-8.

Listing 19-8 - A Custom Configuration Handler, in frontend/lib/myMapConfigHandler.class.php

```
Listing 19-14 <?php

class myMapConfigHandler extends sfYamlConfigHandler
{
    public function execute($configFiles)
    {
        // Parse the yaml
        $config = $this->parseYamls($configFiles);

        $data = "<?php\n";
        $data .= "\$mapApi = new myMapAPI();\n";

        if (isset($config['api']['url']))
        {
            $data .= sprintf("\$mapApi->setUrl('%s');\n", $config['api']['url']);
        }

        if (isset($config['api']['user']))
        {
            $data .= sprintf("\$mapApi->setUser('%s');\n",
                $config['api']['user']);
        }

        return $data;
    }
}
```

The `$configFiles` array that symfony passes to the `execute()` method will contain a path to all the `map.yml` files found in the `config/` folders. The `parseYamls()` method will handle the configuration cascade.

In order to associate this new handler with the `map.yml` file, you must create a `config_handlers.yml` configuration file with the following content:

```
Listing 19-15 config/map.yml:
    class: myMapConfigHandler
```



The `class` must either be autoloaded (that's the case here) or defined in the file whose path is written in a `file` parameter under the `param` key.

As with many other symfony configuration files, you can also register a configuration handler directly in your PHP code:

```
Listing 19-16 sfContext::getInstance()->getConfigCache()->registerConfigHandler('config/
map.yml', 'myMapConfigHandler', array());
```

When you need the code based on the `map.yml` file and generated by the `myMapConfigHandler` handler in your application, call the following line:

```
include sfContext::getInstance()->getConfigCache()->checkConfig('config/  
map.yml');
```

*Listing
19-17*

When calling the `checkConfig()` method, symfony looks for existing `map.yml` files in the configuration directories and processes them with the handler specified in the `config_handlers.yml` file, if a `map.yml.php` does not already exist in the cache or if the `map.yml` file is more recent than the cache.



If you want to handle environments in a YAML configuration file, the handler can extend the `sfDefineEnvironmentConfigHandler` class instead of `sfYamlConfigHandler`. Instead of calling the `parseYaml()` method to retrieve the configuration, you should call the `getConfiguration()` method: `$config = $this->getConfiguration($configFiles)`.

Using Existing configuration handlers

If you just need to allow users to retrieve values from the code via `sfConfig`, you can use the `sfDefineEnvironmentConfigHandler` configuration handler class. For instance, to have the `url` and `user` parameters available as `sfConfig::get('map_url')` and `sfConfig::get('map_user')`, define your handler as follows:

```
config/map.yml:  
  class: sfDefineEnvironmentConfigHandler  
  param:  
    prefix: map_
```

*Listing
19-18*

Be careful not to choose a prefix already used by another handler. Existing prefixes are `sf_`, `app_`, and `mod_`.

Summary

The configuration files can heavily modify the way the framework works. Because symfony relies on configuration even for its core features and file loading, it can adapt to many more environments than just the standard dedicated host. This great configurability is one of the main strengths of symfony. Even if it sometimes frightens newcomers, who see in configuration files a lot of conventions to learn, it allows symfony applications to be compatible with a very large number of platforms and environments. Once you become a master of symfony's configuration, no server will ever refuse to run your applications!

Appendices

Appendix A

Inside The Model Layer (Propel)

Much of the discussion so far has been devoted to building pages, and processing requests and responses. But the business logic of a web application relies mostly on its data model. Symfony's default model component is based on an object/relational mapping layer. Symfony comes bundles with the two most popular PHP ORMs: Propel⁶⁷ and Doctrine⁶⁸. In a symfony application, you access data stored in a database and modify it through objects; you never address the database explicitly. This maintains a high level of abstraction and portability.

This chapter explains how to create an object data model, and the way to access and modify the data in Propel. It also demonstrates the integration of Propel in Symfony.

Why Use an ORM and an Abstraction Layer?

Databases are relational. PHP 5 and symfony are object-oriented. In order to most effectively access the database in an object-oriented context, an interface translating the object logic to the relational logic is required. As explained in Chapter 1, this interface is called an object-relational mapping (ORM), and it is made up of objects that give access to data and keep business rules within themselves.

The main benefit of an ORM is reusability, allowing the methods of a data object to be called from various parts of the application, even from different applications. The ORM layer also encapsulates the data logic—for instance, the calculation of a forum user rating based on how many contributions were made and how popular these contributions are. When a page needs to display such a user rating, it simply calls a method of the data model, without worrying about the details of the calculation. If the calculation changes afterwards, you will just need to modify the rating method in the model, leaving the rest of the application unchanged.

Using objects instead of records, and classes instead of tables, has another benefit: They allow you to add new accessors to your objects that don't necessarily match a column in a table. For instance, if you have a table called `client` with two fields named `first_name` and `last_name`, 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, as in Listing 8-1. From the application point of view, there is no difference between the `FirstName`, `LastName`, and `Name` attributes of the `Client` class. Only the class itself can determine which attributes correspond to a database column.

Listing 8-1 - Accessors Mask the Actual Table Structure in a Model Class

```
public function getName()  
{
```

*Listing
A-1*

67. <http://www.propelorm.org/>

68. <http://www.doctrine-project.org/>

```
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. Suppose you have a `ShoppingCart` class in which you keep `Items` (which are objects). To get the full amount of the shopping cart for the checkout, write a custom method to encapsulate the actual calculation, as shown in Listing 8-2.

Listing 8-2 - Accessors Mask the Data Logic

```
Listing A-2 public function getTotal()
{
    $total = 0;
    foreach ($this->getItems() as $item)
    {
        $total += $item->getPrice() * $item->getQuantity();
    }

    return $total;
}
```

There is another important point to consider when building data-access procedures: Database vendors use different SQL syntax variants. Switching to another database management system (DBMS) forces you to rewrite part of the SQL queries that were designed for the previous one. If you build your queries using a database-independent syntax, and leave the actual SQL translation to a third-party component, you can switch database systems without pain. This is the goal of the database abstraction layer. It forces you to use a specific syntax for queries, and does the dirty job of conforming to the DBMS particulars and optimizing the SQL code.

The main benefit of an abstraction layer is portability, because it makes switching to another database possible, even in the middle of a project. Suppose that you need 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 decide. Just change one line in a configuration file, and it works.

Symfony uses Propel or Doctrine as the ORM, and they use PHP Data Objects for database abstraction. These two third-party components, both developed by the Propel and Doctrine teams, are seamlessly integrated into symfony, and you can consider them as part of the framework. Their syntax and conventions, described in this chapter, were adapted so that they differ from the symfony ones as little as possible.



In a symfony project, all the applications share the same model. That's the whole point of the project level: regrouping applications that rely on common business rules. This is the reason that the model is independent from the applications and the model files are stored in a `lib/model/` directory at the root of the project.

Symfony's Database Schema

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. The ORM needs a description of the relational model to do the mapping, and this is called a schema. In a schema, you define the tables, their relations, and the characteristics of their columns.

Symfony's syntax for schemas uses the YAML format. The `schema.yml` files must be located in the `myproject/config/` directory.

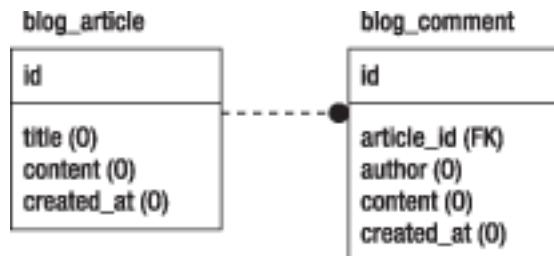


Symfony also understands the Propel native XML schema format, as described in the “Beyond the `schema.yml`: The `schema.xml`” section later in this chapter.

Schema Example

How do you translate a database structure into a schema? An example is the best way to understand it. Imagine that you have a blog database with two tables: `blog_article` and `blog_comment`, with the structure shown in Figure 8-1.

Figure 8-1 - A blog database table structure



The related `schema.yml` file should look like Listing 8-3.

Listing 8-3 - Sample `schema.yml`

```

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

*Listing
A-3*

Notice that the name of the database itself (`blog`) doesn't appear in the `schema.yml` file. 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 in which your application runs. For instance, when you run your application in the development environment, you will access a development database (maybe `blog_dev`), but with the same schema as the production database. The connection settings will be specified in the `databases.yml` file, described in the “Database Connections” section later in this chapter. The schema doesn't contain any detailed connection to settings, only a connection name, to maintain database abstraction.

Basic Schema Syntax

In a `schema.yml` file, the first key represents a connection name. 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` (the name of the class that will be generated). If you don't mention a `phpName` for a table, symfony creates it based on the camelCase version of the table name.



The camelCase convention removes underscores from words, and capitalizes the first letter of inner words. The default camelCase versions of `blog_article` and `blog_comment` are `BlogArticle` and `BlogComment`. The name of this convention comes from the appearance of capitals inside a long word, suggestive of the humps of a camel.

A table contains columns. The column value can be defined in three different ways:

- If you define nothing (~ in YAML is equivalent to `null` in PHP), symfony will guess the best attributes according to the column name and a few conventions that will be described in the "Empty Columns" section later in this chapter. For instance, the `id` column in Listing 8-3 doesn't need to be defined. Symfony will make it an auto-incremented integer, primary key of the table. The `blog_article_id` in the `blog_comment` table will be understood as a foreign key to the `blog_article` table (columns 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. This is one of the reasons why `schema.yml` is so easy to write.
- If you define only one attribute, it is the column type. Symfony understands the usual column types: `boolean`, `integer`, `float`, `date`, `varchar(size)`, `longvarchar` (converted, for instance, to `text` in MySQL), and so on. For text content over 256 characters, you need to use the `longvarchar` type, which has no size (but cannot exceed 65KB in MySQL).
- If you need to define other column attributes (like default value, required, and so on), you should write the column attributes as a set of `key: value`. This extended schema syntax is described later in the chapter.

Columns can also have a `phpName` attribute, which is the capitalized version of the name (`Id`, `Title`, `Content`, and so on) and doesn't need overriding in most cases.

Tables can also contain explicit foreign keys and indexes, as well as a few database-specific structure definitions. Refer to the "Extended Schema Syntax" section later in this chapter to learn more.

Model Classes

The schema is used to build the model classes of the ORM layer. To save execution time, these classes are generated with a command-line task called `propel:build-model`.

Listing A-4 \$ `php symfony propel:build-model`



After building your model, you must remember to clear symfony's internal cache with `php symfony cc` so symfony can find your newly created models.

Typing this command will launch the analysis of the schema and the generation of base data model classes in the `lib/model/om/` directory of your project:

- `BaseArticle.php`
- `BaseArticlePeer.php`
- `BaseComment.php`

- `BaseCommentPeer.php`

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

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

You defined only two tables, and you end up with eight files. There is nothing wrong, but it deserves some explanation.

Base and Custom Classes

Why keep two versions of the data object model in two different directories?

You will probably need to add custom methods and properties to the model objects (think about the `getName()` method in Listing 8-1). But as your project develops, you will also add tables or columns. Whenever you change the `schema.yml` file, you need to regenerate the object model classes by making a new call to `propel:build-model`. If your custom methods were written in the classes actually generated, they would be erased after each generation.

The `Base` classes kept in the `lib/model/om/` directory are the ones directly generated from the schema. You should never modify them, since every new build of the model will completely erase these files.

On the other hand, the custom object classes, kept in the `lib/model/` directory, actually inherit from the `Base` ones. When the `propel:build-model` task is called on an existing model, these classes are not modified. So this is where you can add custom methods.

Listing 8-4 presents an example of a custom model class as created by the first call to the `propel:build-model` task.

Listing 8-4 - Sample Model Class File, in lib/model/Article.php

```
class Article extends BaseArticle
{
}
```

*Listing
A-5*

It inherits all the methods of the `BaseArticle` class, but a modification in the schema will not affect it.

The mechanism of custom classes extending base classes allows you to start coding, even without knowing the final relational model of your database. The related file structure makes the model both customizable and evolutionary.

Object and Peer Classes

`Article` and `Comment` are object classes that represent a record in the database. They give access to the columns of a record and to related records. This means that you will be able to know the title of an article by calling a method of an `Article` object, as in the example shown in Listing 8-5.

Listing 8-5 - Getters for Record Columns Are Available in the Object Class

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

*Listing
A-6*

`ArticlePeer` and `CommentPeer` are peer classes; that is, classes that contain static methods to operate on the tables. They provide a way to retrieve records from the tables.

Their methods usually return an object or a collection of objects of the related object class, as shown in Listing 8-6.

Listing 8-6 - Static Methods to Retrieve Records Are Available in the Peer Class

```
Listing  
A-7 // $articles is an array of objects of class Article  
$articles = ArticlePeer::retrieveByPk(array(123, 124, 125));
```



From a data model point of view, there cannot be any peer object. That's why the methods of the peer classes are called with a `::` (for static method call), instead of the usual `->` (for instance method call).

So combining object and peer classes in a base and a custom version results in four classes generated per table described in the schema. In fact, there is a fifth class created in the `lib/model/map/` directory, which contains metadata information about the table that is needed for the runtime environment. But as you will probably never change this class, you can forget about it.

Accessing Data

In symfony, your data is accessed through objects. If you are used to the relational model and using SQL to retrieve and alter your data, the object model methods will likely look complicated. But once you've tasted the power of object orientation for data access, you will probably like it a lot.

But first, let's make sure we share the same vocabulary. Relational and object data model use similar concepts, but they each have their own nomenclature:

Relational	Object-Oriented
Table	Class
Row, record	Object
Field, column	Property

Retrieving the Column Value

When symfony builds the model, it creates one base object class for each of the tables defined in the `schema.yml`. Each of these classes comes with default constructors, accessors, and mutators based on the column definitions: The `new`, `getXXX()`, and `setXXX()` methods help to create objects and give access to the object properties, as shown in Listing 8-7.

Listing 8-7 - Generated Object Class Methods

```
Listing  
A-8 $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();
```



The generated object class is called `Article`, which is the `phpName` given to the `blog_article` table. If the `phpName` were not defined in the schema, the class would have

been called `BlogArticle`. The accessors and mutators use a camelCase variant of the column names, so the `getTitle()` method retrieves the value of the `title` column.

To set several fields at one time, you can use the `fromArray()` method, also generated for each object class, as shown in Listing 8-8.

Listing 8-8 - The `fromArray()` Method Is a Multiple Setter

```
$article->fromArray(array(
    'Title'    => 'My first article',
    'Content'  => 'This is my very first article.\n Hope you enjoy it!',
));
```

*Listing
A-9*



The `fromArray()` method has a second argument `keyType`. You can specify the key type of the array by additionally passing one of the class type constants `BasePeer::TYPE_PHPNAME`, `BasePeer::TYPE_STUDLYPHPNAME`, `BasePeer::TYPE_COLNAME`, `BasePeer::TYPE_FIELDNAME`, `BasePeer::TYPE_NUM`. The default key type is the column's PhpName (e.g. 'AuthorId').

Retrieving Related Records

The `blog_article_id` column in the `blog_comment` table implicitly defines a foreign key to the `blog_article` table. Each comment is related to one article, and one article can have many comments. The generated classes contain five methods translating this relationship in an object-oriented way, as follows:

- `$comment->getArticle()`: To get the related `Article` object
- `$comment->getArticleId()`: To get the ID of the related `Article` object
- `$comment->setArticle($article)`: To define the related `Article` object
- `$comment->setArticleId($id)`: To define the related `Article` object from an ID
- `$article->getComments()`: To get the related `Comment` objects

The `getArticleId()` and `setArticleId()` methods show that you can consider the `blog_article_id` column as a regular column and set the relationships by hand, but they are not very interesting. The benefit of the object-oriented approach is much more apparent in the three other methods. Listing 8-9 shows how to use the generated setters.

Listing 8-9 - Foreign Keys Are Translated into a Special Setter

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

// Attach this comment to the previous $article object
$comment->setArticle($article);

// Alternative syntax
// Only makes sense if the object is already saved in the database
$comment->setArticleId($article->getId());
```

*Listing
A-10*

Listing 8-10 shows how to use the generated getters. It also demonstrates how to chain method calls on model objects.

Listing 8-10 - Foreign Keys Are Translated into Special Getters

Listing A-11

```
// Many to one relationship
echo $comment->getArticle()->getTitle();
=> My first article
echo $comment->getArticle()->getContent();
=> This is my very first article.
    Hope you enjoy it!

// One to many relationship
$comments = $article->getComments();
```

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).

The `$comments` variable in Listing 8-10 contains an array of objects of class `Comment`. You can display the first one with `$comments[0]` or iterate through the collection with `foreach ($comments as $comment)`.



Objects from the model are defined with a singular name by convention, and you can now understand why. The foreign key defined in the `blog_comment` table causes the creation of a `getComments()` method, named by adding an `s` to the `Comment` object name. If you gave the model object a plural name, the generation would lead to a method named `getCommentss()`, which doesn't make sense.

Saving and Deleting Data

By calling the `new` constructor, you created a new object, but not an actual record in the `blog_article` table. Modifying the object has no effect on the database either. In order to save the data into the database, you need to call the `save()` method of the object.

Listing A-12

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

The ORM is smart enough to detect relationships between objects, so saving the `$article` object also saves the related `$comment` object. It also knows if the saved object has an existing counterpart in the database, so the call to `save()` is sometimes translated in SQL by an `INSERT`, and sometimes by an `UPDATE`. The primary key is automatically set by the `save()` method, so after saving, you can retrieve the new primary key with `$article->getId()`.



You can check if an object is new by calling `isNew()`. And if you wonder if an object has been modified and deserves saving, call its `isModified()` method.

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, as shown in Listing 8-11.

Listing 8-11 - Delete Records from the Database with the delete() Method on the Related Object

Listing A-13

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



Even after calling the `delete()` method, an object remains available until the end of the request. To determine if an object is deleted in the database, call the `isDeleted()` method.

Retrieving Records by Primary Key

If you know the primary key of a particular record, use the `retrieveByPk()` class method of the peer class to get the related object.

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

Listing A-14

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

In some cases, a primary key may consist of more than one column. In those cases, the `retrieveByPK()` method accepts multiple parameters, one for each primary key column.

You can also select multiple objects based on their primary keys, by calling the generated `retrieveByPKs()` method, which expects an array of primary keys as a parameter.

Retrieving Records with Criteria

When you want to retrieve more than one record, you need to call the `doSelect()` method of the peer class corresponding to the objects you want to retrieve. For instance, to retrieve objects of class `Article`, call `ArticlePeer::doSelect()`.

The first parameter of the `doSelect()` method is an object of class `Criteria`, which is a simple query definition class defined without SQL for the sake of database abstraction.

An empty `Criteria` returns all the objects of the class. For instance, the code shown in Listing 8-12 retrieves all the articles.

Listing 8-12 - Retrieving Records by Criteria with doSelect()—Empty Criteria

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

// Will result in the following SQL query
SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT,
       blog_article.CREATED_AT
FROM   blog_article;
```

Listing A-15

Hydrating

The call to `::doSelect()` is actually much more powerful than a simple SQL query. First, the SQL is optimized for the DBMS you choose. Second, any value passed to the `Criteria` is escaped before being integrated into the SQL code, which prevents SQL injection risks. Third, the method returns an array of objects, rather than a result set. The ORM automatically creates and populates objects based on the database result set. This process is called hydrating.

For a more complex object selection, you need an equivalent of the WHERE, ORDER BY, GROUP BY, and other SQL statements. The `Criteria` object has methods and parameters for all these conditions. For example, to get all comments written by Steve, ordered by date, build a `Criteria` as shown in Listing 8-13.

Listing 8-13 - Retrieving Records by Criteria with doSelect()—Criteria with Conditions

Listing A-16

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

// Will result in the following SQL query
SELECT blog_comment.ARTICLE_ID, blog_comment.AUTHOR, blog_comment.CONTENT,
       blog_comment.CREATED_AT
FROM   blog_comment
WHERE  blog_comment.author = 'Steve'
ORDER BY blog_comment.CREATED_AT ASC;
```

The class constants passed as parameters to the `add()` methods refer to the property names. They are named after the capitalized version of the column names. For instance, to address the `content` column of the `blog_article` table, use the `ArticlePeer::CONTENT` class constant.



Why use `CommentPeer::AUTHOR` instead of `blog_comment.AUTHOR`, which is the way it will be output in the SQL query anyway? Suppose that you need to change the name of the author field to `contributor` in the database. If you used `blog_comment.AUTHOR`, you would have to change it in every call to the model. On the other hand, by using `CommentPeer::AUTHOR`, you simply need to change the column name in the `schema.yml` file, keep `phpName` as `AUTHOR`, and rebuild the model.

Table 8-1 compares the SQL syntax with the `Criteria` object syntax.

Table 8-1 - SQL and Criteria Object Syntax

SQL	Criteria
<code>WHERE column = value</code>	<code>->add(column, value);</code>
<code>WHERE column <> value</code>	<code>->add(column, value, Criteria::NOT_EQUAL);</code>
Other Comparison Operators	
<code>> , <</code>	<code>Criteria::GREATER_THAN, Criteria::LESS_THAN</code>
<code>>=, <=</code>	<code>Criteria::GREATER_EQUAL, Criteria::LESS_EQUAL</code>
<code>IS NULL, IS NOT NULL</code>	<code>Criteria::ISNULL, Criteria::ISNOTNULL</code>
<code>LIKE, ILIKE</code>	<code>Criteria::LIKE, Criteria::ILIKE</code>
<code>IN, NOT IN</code>	<code>Criteria::IN, Criteria::NOT_IN</code>
Other SQL Keywords	
<code>ORDER BY column ASC</code>	<code>->addAscendingOrderByColumn(column);</code>
<code>ORDER BY column DESC</code>	<code>- >addDescendingOrderByColumn(column);</code>
<code>LIMIT limit</code>	<code>->setLimit(limit)</code>
<code>OFFSET offset</code>	<code>->setOffset(offset)</code>
<code>FROM table1, table2 WHERE table1.col1 = table2.col2</code>	<code>->addJoin(col1, col2)</code>

SQL	Criteria
FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::LEFT_JOIN)
FROM table1 RIGHT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::RIGHT_JOIN)



The best way to discover and understand which methods are available in generated classes is to look at the `Base` files in the `lib/model/om/` folder after generation. The method names are pretty explicit, but if you need more comments on them, set the `propel.builder.addComments` parameter to true in the `config/propel.ini` file and rebuild the model.

Listing 8-14 shows another example of `Criteria` with multiple conditions. It retrieves all the comments by Steve on articles containing the word “enjoy,” ordered by date.

Listing 8-14 - Another Example of Retrieving Records by Criteria with doSelect()—Criteria with Conditions

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addJoin(CommentPeer::ARTICLE_ID, ArticlePeer::ID);
$c->add(ArticlePeer::CONTENT, '%enjoy%', Criteria::LIKE);
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);

// Will result in the following SQL query
SELECT blog_comment.ID, blog_comment.ARTICLE_ID, blog_comment.AUTHOR,
       blog_comment.CONTENT, blog_comment.CREATED_AT
FROM   blog_comment, blog_article
WHERE  blog_comment.AUTHOR = 'Steve'
       AND blog_article.CONTENT LIKE '%enjoy%'
       AND blog_comment.ARTICLE_ID = blog_article.ID
ORDER BY blog_comment.CREATED_AT ASC
```

Listing A-17

Just as SQL is a simple language that allows you to build very complex queries, the `Criteria` object can handle conditions with any level of complexity. But since many developers think first in SQL before translating a condition into object-oriented logic, the `Criteria` object may be difficult to comprehend at first. The best way to understand it is to learn from examples and sample applications. The `symfony` project website, for instance, is full of `Criteria` building examples that will enlighten you in many ways.

In addition to the `doSelect()` method, every peer class has a `doCount()` method, which simply counts the number of records satisfying the criteria passed as a parameter and returns the count as an integer. As there is no object to return, the hydrating process doesn't occur in this case, and the `doCount()` method is faster than `doSelect()`.

The peer classes also provide `doDelete()`, `doInsert()`, and `doUpdate()` methods, which all expect a `Criteria` as a parameter. These methods allow you to issue `DELETE`, `INSERT`, and `UPDATE` queries to your database. Check the generated peer classes in your model for more details on these Propel methods.

Finally, if you just want the first object returned, replace `doSelect()` with a `doSelectOne()` call. This may be the case when you know that a `Criteria` will return only one result, and the advantage is that this method returns an object rather than an array of objects.



When a `doSelect()` query returns a large number of results, you might want to display only a subset of it in your response. Symfony provides a pager class called `sFPropelPager`, which automates the pagination of results.

Using Raw SQL Queries

Sometimes, you don't want to retrieve objects, but want to get only synthetic results calculated by the database. For instance, to get the latest creation date of all articles, it doesn't make sense to retrieve all the articles and to loop on the array. You will prefer to ask the database to return only the result, because it will skip the object hydrating process.

On the other hand, you don't want to call the PHP commands for database management directly, because then you would lose the benefit of database abstraction. This means that you need to bypass the ORM (Propel) but not the database abstraction (PDO).

Querying the database with PHP Data Objects requires that you do the following:

1. Get a database connection.
2. Build a query string.
3. Create a statement out of it.
4. Iterate on the result set that results from the statement execution.

If this looks like gibberish to you, the code in Listing 8-15 will probably be more explicit.

Listing 8-15 - Custom SQL Query with PDO

Listing A-18

```
$connection = Propel::getConnection();
$query = 'SELECT MAX(?) AS max FROM ?';
$statement = $connection->prepare($query);
$statement->bindValue(1, ArticlePeer::CREATED_AT);
$statement->bindValue(2, ArticlePeer::TABLE_NAME);
$statement->execute();
$resultset = $statement->fetch(PDO::FETCH_OBJ);
$max = $resultset->max;
```

Just like Propel selections, PDO queries are tricky when you first start using them. Once again, examples from existing applications and tutorials will show you the right way.



If you are tempted to bypass this process and access the database directly, you risk losing the security and abstraction provided by Propel. Doing it the Propel way is longer, but it forces you to use good practices that guarantee the performance, portability, and security of your application. This is especially true for queries that contain parameters coming from a untrusted source (such as an Internet user). Propel does all the necessary escaping and secures your database. Accessing the database directly puts you at risk of SQL-injection attacks.

Using Special Date Columns

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

The good news is that Symfony will recognize the names of these columns and handle their updates for you. You don't need to manually set the `created_at` and `updated_at` columns; they will automatically be updated, as shown in Listing 8-16. The same applies for columns named `created_on` and `updated_on`.

Listing 8-16 - `created_at` and `updated_at` Columns Are Dealt with Automatically

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->save();

// Show the creation date
echo $comment->getCreatedAt();
=> [date of the database INSERT operation]
```

*Listing
A-19*

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

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

*Listing
A-20*

Refactoring to the Data layer

When developing a symfony project, you often start by writing the domain logic code in the actions. But the database queries and model manipulation should not be stored in the controller layer. So 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 short and readable.

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

```
public function executeShowPopularArticlesForTag($request)
{
    $tag = TagPeer::retrieveByName($request->getParameter('tag'));
    $this->forward404Unless($tag);
    $this->articles = $tag->getPopularArticles(10);
}
```

*Listing
A-21*

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. It makes the action more readable, and the model code can easily be reused in another action.

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 contain more than ten lines of PHP code.

Database Connections

The data model is independent from the database used, but you will definitely use a database. The minimum information required by symfony to send requests to the project database is the name, the credentials, and the type of database. These connection settings can be configured by passing a data source name (DSN) to the `configure:database` task:

```
$ php symfony configure:database "mysql:host=localhost;dbname=blog" root
mYsEcret
```

*Listing
A-22*

The connection settings are environment-dependent. You can define distinct settings for the `prod`, `dev`, and `test` environments, or any other environment in your application by using the `env` option:

Listing A-23

```
$ php symfony configure:database --env=dev
"mysql:host=localhost;dbname=blog_dev" root mYsEcret
```

This configuration can also be overridden per application. For instance, you can use this approach to have different security policies for a front-end and a back-end application, and define several database users with different privileges in your database to handle this:

Listing A-24

```
$ php symfony configure:database --app=frontend
"mysql:host=localhost;dbname=blog" root mYsEcret
```

For each environment, you can define many connections. Each connection refers to a schema being labeled with the same name. The default connection name used is `propel` and it refers to the `propel` schema in Listing 8-3. The `name` option allows you to create another connection:

Listing A-25

```
$ php symfony configure:database --name=main
"mysql:host=localhost;dbname=example" root mYsEcret
```

You can also enter these connection settings manually in the `databases.yml` file located in the `config/` directory. Listing 8-17 shows an example of such a file and Listing 8-18 shows the same example with the extended notation.

Listing 8-17 - Shorthand Database Connection Settings

Listing A-26

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

Listing 8-18 - Sample Database Connection Settings, in myproject/config/databases.yml

Listing A-27

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

all:
propel:
  class:          sfPropelDatabase
  param:
    phptype:       mysql      # Database vendor
    hostspec:      localhost
    database:      blog
    username:      login
    password:      passwd
    port:          80
    encoding:      utf8       # Default charset for table creation
    persistent:    true       # Use persistent connections
```

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

- mysql
- mssql

- `pgsql`
- `sqlite`
- `oracle`

`hostspec`, `database`, `username`, and `password` are the usual database connection settings.

To override the configuration per application, you need to edit an application-specific file, such as `apps/frontend/config/databases.yml`.

If you use a SQLite database, the `hostspec` parameter must be set to the path of the database file. For instance, if you keep your blog database in `data/blog.db`, the `databases.yml` file will look like Listing 8-19.

Listing 8-19 - Database Connection Settings for SQLite Use a File Path As Host

```
all:
  propel:
    class:      sfPropelDatabase
    param:
      phptype:  sqlite
      database: %SF_DATA_DIR%/blog.db
```

*Listing
A-28*

Extending the Model

The generated model methods are great but often not sufficient. As soon as you implement your own business logic, you need to extend it, either by adding new methods or by overriding existing ones.

Adding New Methods

You can add new methods to the empty model classes generated in the `lib/model/` directory. Use `$this` to call methods of the current object, and use `self::` to call static methods of the current class. Remember that the custom classes inherit methods from the `Base` classes located in the `lib/model/om/` directory.

For instance, for the `Article` object generated based on Listing 8-3, you can add a magic `__toString()` method so that echoing an object of class `Article` displays its title, as shown in Listing 8-20.

Listing 8-20 - Customizing the Model, in lib/model/Article.php

```
class Article extends BaseArticle
{
  public function __toString()
  {
    return $this->getTitle(); // getTitle() is inherited from BaseArticle
  }
}
```

*Listing
A-29*

You can also extend the peer classes—for instance, to add a method to retrieve all articles ordered by creation date, as shown in Listing 8-21.

Listing 8-21 - Customizing the Model, in lib/model/ArticlePeer.php

```
class ArticlePeer extends BaseArticlePeer
{
  public static function getAllOrderedByDate()
  {
    $c = new Criteria();

```

*Listing
A-30*

```

$c->addAscendingOrderByColumn(self::CREATED_AT);

return self::doSelect($c);
}
}

```

The new methods are available in the same way as the generated ones, as shown in Listing 8-22.

Listing 8-22 - Using Custom Model Methods Is Like Using the Generated Methods

```

Listing A-31 foreach (ArticlePeer::getAllOrderedByDate() as $article)
{
    echo $article;      // Will call the magic __toString() method
}

```

Overriding Existing Methods

If some of the generated methods in the `Base` classes don't fit your requirements, you can still override them in the custom classes. Just make sure that you use the same method signature (that is, the same number of arguments).

For instance, the `$article->getComments()` method returns an array of `Comment` objects, in no particular order. If you want to have the results ordered by creation date, with the latest comment coming first, then override the `getComments()` method, as shown in Listing 8-23. Be aware that the original `getComments()` method (found in `lib/model/om/BaseArticle.php`) expects a criteria value and a connection value as parameters, so your function must do the same.

Listing 8-23 - Overriding Existing Model Methods, in lib/model/Article.php

```

Listing A-32 public function getComments($criteria = null, $con = null)
{
    if (is_null($criteria))
    {
        $criteria = new Criteria();
    }
    else
    {
        // Objects are passed by reference in PHP5, so to avoid modifying the
        // original, you must clone it
        $criteria = clone $criteria;
    }
    $criteria->addDescendingOrderByColumn(CommentPeer::CREATED_AT);

    return parent::getComments($criteria, $con);
}

```

The custom method eventually calls the one of the parent `Base` class, and that's good practice. However, you can completely bypass it and return the result you want.

Using Model Behaviors

Some model modifications are generic and can be reused. For instance, methods to make a model object sortable and an optimistic lock to prevent conflicts between concurrent object saving are generic extensions that can be added to many classes.

Symfony packages these extensions into behaviors. Behaviors are external classes that provide additional methods to model classes. The model classes already contain hooks, and symfony knows how to extend them.

To enable behaviors in your model classes, you must modify one setting in the `config/propel.ini` file:

```
propel.builder.AddBehaviors = true      // Default value is false
```

Listing A-33

There is no behavior bundled by default in symfony, but they can be installed via plug-ins. Once a behavior plug-in is installed, you can assign the behavior to a class with a single line. For instance, if you install the `sfPropelParanoidBehaviorPlugin` in your application, you can extend an `Article` class with this behavior by adding the following at the end of the `Article.class.php`:

```
sfPropelBehavior::add('Article', array(
    'paranoid' => array('column' => 'deleted_at')
));
```

Listing A-34

After rebuilding the model, deleted `Article` objects will remain in the database, invisible to the queries using the ORM, unless you temporarily disable the behavior with `sfPropelParanoidBehavior::disable()`.

Alternatively, you can also declare behaviors directly in the `schema.yml`, by listing them under the `_behaviors` key (see Listing 8-34 below).

Check the list of symfony plug-ins on the official repository⁶⁹ to find behaviors. Each has its own documentation and installation guide.

Extended Schema Syntax

A `schema.yml` file can be simple, as shown in Listing 8-3. But relational models are often complex. That's why the schema has an extensive syntax able to handle almost every case.

Attributes

Connections and tables can have specific attributes, as shown in Listing 8-24. They are set under an `_attributes` key.

Listing 8-24 - Attributes for Connections and Tables

```
propel:
  _attributes: { noXsd: false, defaultIdMethod: none, package: lib.model }
}
blog_article:
  _attributes: { phpName: Article }
```

Listing A-35

You may want your schema to be validated before code generation takes place. To do that, deactivate the `noXSD` attribute for the connection. The connection also supports the `defaultIdMethod` attribute. If `none` is provided, then the database's native method of generating IDs will be used—for example, `autoincrement` for MySQL, or `sequences` for PostgreSQL. The other possible value is `none`.

The `package` attribute is like a namespace; it determines the path where the generated classes are stored. It defaults to `lib/model/`, but you can change it to organize your model in subpackages. For instance, if you don't want to mix the core business classes and the

69. <http://www.symfony-project.org/plugins/>

classes defining a database-stored statistics engine in the same directory, then define two schemas with `lib.model.business` and `lib.model.stats` packages.

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

Tables that contain localized content (that is, several versions of the content, in a related table, for internationalization) also take two additional attributes (see Chapter 13 for details), as shown in Listing 8-25.

Listing 8-25 - Attributes for i18n Tables

Listing
A-36 `propel:`
 `blog_article:`
 `_attributes: { isI18N: true, i18nTable: db_group_i18n }`

Dealing with multiple Schemas

You can have more than one schema per application. Symfony will take into account every file ending with `schema.yml` or `schema.xml` in the `config/` folder. If your application has many tables, or if some tables don't share the same connection, you will find this approach very useful.

Consider these two schemas:

```
// In config/business-schema.yml
propel:
    blog_article:
        _attributes: { phpName: Article }
        id:
        title: varchar(50)

// In config/stats-schema.yml
propel:
    stats_hit:
        _attributes: { phpName: Hit }
        id:
        resource: varchar(100)
        created_at:
```

*Listing
A-37*

Both schemas share the same connection (`propel`), and the `Article` and `Hit` classes will be generated under the same `lib/model/` directory. Everything happens as if you had written only one schema.

You can also have different schemas use different connections (for instance, `propel` and `propel_bis`, to be defined in `databases.yml`) and organize the generated classes in subdirectories:

```
// In config/business-schema.yml
propel:
    blog_article:
        _attributes: { phpName: Article, package: lib.model.business }
        id:
        title: varchar(50)

// In config/stats-schema.yml
propel_bis:
    stats_hit:
        _attributes: { phpName: Hit, package: lib.model.stat }
        id:
        resource: varchar(100)
        created_at:
```

*Listing
A-38*

Many applications use more than one schema. In particular, some plug-ins have their own schema and package to avoid messing with your own classes (see Chapter 17 for details).

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. Listing 8-26 demonstrates these choices.

Listing 8-26 - Basic Column Attributes

*Listing
A-39*

```
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. If you do, you will need to define column settings as an associative array, as shown in Listing 8-27.

Listing 8-27 - Complex Column Attributes

Listing A-40

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

The column parameters are as follows:

- **type**: Column type. The choices are `boolean`, `tinyint`, `smallint`, `integer`, `bigint`, `double`, `float`, `real`, `decimal`, `char`, `varchar(size)`, `longvarchar`, `date`, `time`, `timestamp`, `bu_date`, `bu_timestamp`, `blob`, and `clob`.
- **required**: Boolean. Set it to `true` if you want the column to be required.
- **size**: The size or length of the field for types that support it
- **scale**: Number of decimal places for use with decimal data type (size must also be specified)
- **default**: Default value.
- **primaryKey**: Boolean. Set it to `true` for primary keys.
- **autoIncrement**: Boolean. Set it to `true` for columns of type `integer` that need to take an auto-incremented value.
- **sequence**: Sequence name for databases using sequences for `autoIncrement` columns (for example, PostgreSQL and Oracle).
- **index**: Boolean. Set it to `true` if you want a simple index or to `unique` if you want a unique index to be created on the column.
- **foreignTable**: A table name, used to create a foreign key to another table.
- **foreignReference**: The name of the related column if a foreign key is defined via `foreignTable`.
- **onDelete**: Determines the action to trigger when a record in a related table is deleted. When set to `setnull`, the foreign key column is set to `null`. When set to `cascade`, the record is deleted. If the database engine doesn't support the `set` behavior, the ORM emulates it. This is relevant only for columns bearing a `foreignTable` and a `foreignReference`.
- **isCulture**: Boolean. Set it to `true` for culture columns in localized content tables (see Chapter 13).

Foreign Keys

As an alternative to the `foreignTable` and `foreignReference` column attributes, you can add foreign keys under the `_foreignKeys:` key in a table. The schema in Listing 8-28 will create a foreign key on the `user_id` column, matching the `id` column in the `blog_user` table.

Listing 8-28 - Foreign Key Alternative Syntax

Listing A-41

```
propel:
  blog_article:
```

```

id:      ~
title:   varchar(50)
user_id: { type: integer }
_FOREIGNKEYS:
-
  foreignTable: blog_user
  onDelete:    cascade
  references:
    - { local: user_id, foreign: id }

```

The alternative syntax is useful for multiple-reference foreign keys and to give foreign keys a name, as shown in Listing 8-29.

Listing 8-29 - Foreign Key Alternative Syntax Applied to Multiple Reference Foreign Key

```

_FOREIGNKEYS:
my_foreign_key:
  foreignTable: db_user
  onDelete:    cascade
  references:
    - { local: user_id, foreign: id }
    - { local: post_id, foreign: id }

```

*Listing
A-42*

Indexes

As an alternative to the `index` column attribute, you can add indexes under the `_indexes`: key in a table. If you want to define unique indexes, you must use the `_uniques`: header instead. For columns that require a size, because they are text columns, the size of the index is specified the same way as the length of the column using parentheses. Listing 8-30 shows the alternative syntax for indexes.

Listing 8-30 - Indexes and Unique Indexes Alternative Syntax

```

propel:
  blog_article:
    id:          ~
    title:       varchar(50)
    created_at:
    _indexes:
      my_index: [title(10), user_id]
    _uniques:
      my_other_index: [created_at]

```

*Listing
A-43*

The alternative syntax is useful only for indexes built on more than one column.

Empty Columns

When meeting a column with no value, symfony will do some magic and add a value of its own. See Listing 8-31 for the details added to empty columns.

Listing 8-31 - Column Details Deduced from the Column Name

```

// Empty columns named id are considered primary keys
id: { type: integer, required: true, primaryKey: true,
autoIncrement: true }

// Empty columns named XXX_id are considered foreign keys
foobar_id: { type: integer, foreignTable: db_foobar, foreignReference: id }

```

*Listing
A-44*

```
// Empty columns named created_at, updated_at, created_on and updated_on
// are considered dates and automatically take the timestamp type
created_at: { type: timestamp }
updated_at: { type: timestamp }
```

For foreign keys, 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`.

I18n Tables

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

In a `schema.yml` file, all that is implied when you name a table `foobar_i18n`. For instance, the schema shown in Listing 8-32 will be automatically completed with columns and table attributes to make the internationalized content mechanism work. Internally, symfony will understand it as if it were written like Listing 8-33. Chapter 13 will tell you more about i18n.

Listing 8-32 - Implied i18n Mechanism

```
Listing A-45 propel:
  db_group:
    id: ~
    created_at: ~

  db_group_i18n:
    name: varchar(50)
```

Listing 8-33 - Explicit i18n Mechanism

```
Listing A-46 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, foreignReference: id, onDelete: cascade }
    culture: { isCulture: true, type: varchar(7), required: true, primaryKey: true }
    name: varchar(50)
```

Behaviors

Behaviors are model modifiers provided by plug-ins that add new capabilities to your Propel classes. Chapter 17 explains more about behaviors. You can define behaviors right in the schema, by listing them for each table, together with their parameters, under the `_behaviors` key. Listing 8-34 gives an example by extending the `BlogArticle` class with the `paranoid` behavior.

Listing 8-34 - Behaviors Declaration

```
Listing A-47 propel:
  blog_article:
    title: varchar(50)
```

```
_behaviors:
  paranoid: { column: deleted_at }
```

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` file contains the same information as its YAML equivalent. For example, Listing 8-3 is converted to the XML file shown in Listing 8-35.

Listing 8-35 - Sample schema.xml, Corresponding to Listing 8-3

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="propel" defaultIdMethod="native" noXsd="true"
package="lib.model">
  <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>
```

*Listing
A-48*

The description of the `schema.xml` format can be found in the documentation and the “Getting Started” sections of the Propel project website⁷⁰.

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

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 XML syntax, you don't have to switch to the symfony YAML syntax. Place your `schema.xml` in the project `config/` directory, build the model, and there you go.

70. http://propel.phpdb.org/docs/user_guide/chapters/applications/AppendixB-SchemaReference.html

Propel in symfony

All the details given in this chapter are not specific to symfony, but rather to Propel. Propel is the preferred object/relational abstraction layer for symfony, but you can choose an alternative one. However, symfony works more seamlessly with Propel, for the following reasons:

All the object data model classes and the `Criteria` class are autoloading classes. As soon as you use them, symfony will include the right files, and you don't need to manually add the file inclusion statements. In symfony, Propel doesn't need to be launched nor initialized. When an object uses Propel, the library initiates by itself. Some symfony helpers use Propel objects as parameters to achieve high-level tasks (such as pagination or filtering). Propel objects allow rapid prototyping and generation of a backend for your application (Chapter 14 provides more details). The schema is faster to write through the `schema.yml` file.

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

Don't Create the Model Twice

The trade-off of using an ORM is that you must define the data structure twice: once for the database, and once for the object model. Fortunately, symfony offers command-line tools to generate one based on the other, so you can avoid duplicate work.

Building a SQL Database Structure Based on an Existing Schema

If you start your application by writing the `schema.yml` file, symfony can generate a SQL query that creates the tables directly from the YAML data model. To use the query, go to your root project directory and type this:

Listing A-49 \$ `php symfony propel:build-sql`

A `lib.model.schema.sql` file 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 `propel.ini` file.

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

Listing A-50 \$ `mysqladmin -u root -p create blog`
\$ `mysql -u root -p blog < data/sql/lib.model.schema.sql`

The generated SQL is also helpful to rebuild the database in another environment, or to change to another DBMS. If the connection settings are properly defined in your `propel.ini`, you can even use the `php symfony propel:insert-sql` command to do this automatically.



The command line also offers a task to populate your database with data based on a text file. See Chapter 16 for more information about the `propel:data-load` task and the YAML fixture files.

Generating a YAML Data Model from an Existing Database

Symfony can use Propel to generate a `schema.yml` file from an existing database, thanks to introspection (the capability of databases to determine the structure of the tables on which

they are operating). 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 need to make sure that the project `databases.yml` file points to the correct database and contains all connection settings, and then call the `propel:build-schema` command:

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

Listing A-51

A brand-new `schema.yml` file built from your database structure is generated in the `config/` directory. You can build your model based on this schema.

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

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

Listing A-52

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

The `propel.ini` Configuration

This file contains other settings used to configure the Propel generator to make generated model classes compatible with symfony. Most settings are internal and of no interest to the user, apart from a few:

```
// Base classes are autoloaded in symfony
// Set this to true to use include_once statements instead
// (Small negative impact on performance)
propel.builder.addIncludes = false

// Generated classes are not commented by default
// Set this to true to add comments to Base classes
// (Small negative impact on performance)
propel.builder.addComments = false

// Behaviors are not handled by default
// Set this to true to be able to handle them
propel.builder.AddBehaviors = false
```

Listing A-53

After you make a modification to the `propel.ini` settings, don't forget to rebuild the model so the changes will take effect.

Summary

Symfony uses Propel as the ORM and PHP Data Objects as the database abstraction layer. It means that you must first describe the relational schema of your database in YAML before generating the object model classes. Then, at runtime, use the methods of the object and peer classes to retrieve information about a record or a set of records. You can override them and extend the model easily by adding methods to the custom classes. The connection settings are defined in a `databases.yml` file, which can support more than one connection. And the command line contains special tasks to avoid duplicate structure definition.

The model layer is the most complex of the symfony framework. One reason for this complexity is that data manipulation is an intricate matter. The related security issues are crucial for a website and should not be ignored. Another reason is that symfony is more suited for middle- to large-scale applications in an enterprise context. In such applications, the automations provided by the symfony model really represent a gain of time, worth the investment in learning its internals.

So don't hesitate to spend some time testing the model objects and methods to fully understand them. The solidity and scalability of your applications will be a great reward.

Appendix B

GNU Free Documentation License

- Version 1.2, November 2002
- Copyright © 2000, 2001, 2002 Free Software Foundation, Inc.
- 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a worldwide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be

listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are

acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>⁷¹.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

⁷¹. <http://www.gnu.org/copyleft/>



We publish Open-Source Books for demanding People

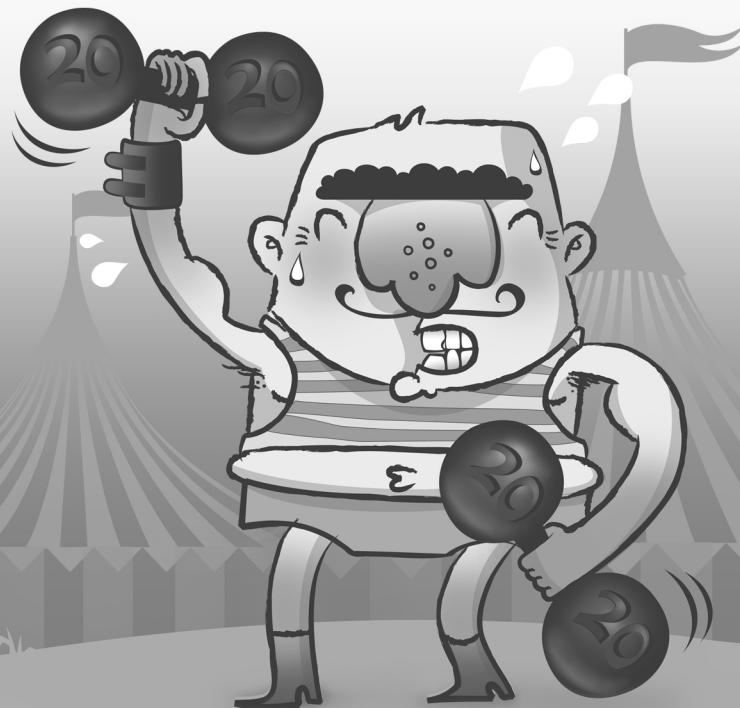
Learn from Open-Source experts



Discover more titles on
<http://books.sensiolabs.com/>

Learn more About Open-Source technologies

Be trained by Open-Source specialists



Visit today **<http://trainings.sensiolabs.com/>**
and save 10% on your next training

Coupon code

BOOKS305

Offer valid through 09/30/2010

