# More with symfony

symfony 1.3 & 1.4

This PDF is brought to you by

SENSIOLABS


*Version*: more-with-symfony-1.4-en-2012-10-17

# Table of Contents

# About the Authors

## Fabien Potencier

Fabien Potencier discovered the Web in 1994, at a time when connecting to the Internet was still associated with the harmful strident sounds of a modem. Being a developer by passion, he immediately started to build websites with Perl. But with the release of PHP 5, he decided to switch focus to PHP, and created the symfony[1] framework project in 2004 to help his company leverage the power of PHP for its customers.

Fabien is a serial-entrepreneur, and among other companies, he created Sensio[2], a services and consulting company specialized in web technologies and Internet marketing, in 1998.

Fabien is also the creator of several other Open-Source projects, a writer, a blogger, a speaker at international conferences, and a happy father of two wonderful kids.

**His Website**: *http://fabien.potencier.org/*

**On Twitter**: *http://twitter.com/fabpot*

## Jonathan H. Wage

Jonathan Wage has been working with various web technologies for nearly 10 years building software on the internet. He formerly was the lead developer and application architect at CentreSource[3]. Today you will find him with Sensio Labs[4], the creators of the symfony MVC framework. His primary responsibilities are training other developers as well as working on the symfony and Doctrine open source projects. He is a core contributor to symfony and the lead of the Doctrine project.

**His Website**: *http://www.jwage.com/*

**On Twitter**: *http://twitter.com/jwage*

## Geoffrey Bachelet

Geoffrey is a self-made developer building web applications since the early 2000s. Curious about new technologies and always trying to learn new things, he tried a variety of web-related technologies, including (but not limited to) various frameworks such as Zend Framework, Ruby on Rails and merb. He finally settled at Sensio Labs[5] where he developed a

1. http://www.symfony-project.org/
2. http://www.sensio.com/
3. http://www.centresource.com
4. http://www.sensiolabs.com
5. http://www.sensiolabs.com/

great understanding of the symfony PHP framework. He is also a founding member of the yet-to-be officially announced french-speaking symfony users association, also known as AFSY, and is looking forward to getting involved more deeply in the symfony community by writing and giving talks about symfony.

**His Website**: *http://mirmodynamics.com/*

**On Twitter**: *http://twitter.com/ubermuda*

## Kris Wallsmith

Kris Wallsmith is a freelance web developer from Portland, Oregon. He spends his days in the Silicon Forrest leading the initial builds of web startups, playing soccer and raising his two young children. Kris is also speaker at international conferences, member of the symfony core team and Release Manager for versions 1.3 and 1.4.

**His Website**: *http://kriswallsmith.net*

**On Twitter**: *http://twitter.com/kriswallsmith*

## Hugo Hamon

Hugo Hamon has been a web technologies autodidact since late 2000 and now works at Sensio Labs as a web developer and training manager. Passionate about PHP, he has used it for 8 years and built the French tutorial website Apprendre-PHP.com[6]. Hugo is involved in two french associations: the AFUP[7] as a staff member and the AFSY[8], the symfony users group, as the founder and President. He's the co-author of the book *Mieux Programmer en PHP avec symfony 1.2 et Doctrine* at Eyrolles publishing[9] and other articles for the magazine PHP Solutions.

**His Website**: *http://www.hugohamon.com*

**On Twitter**: *http://twitter.com/hhamon*

## Thomas Rabaix

Thomas has been a symfony fan since the early days. He is currently working as a PHP and symfony freelancer (developer, consultant and trainer) and is very active in the symfony community.

Thomas also helps companies with Quality Assurance and Architecture questions on challenging symfony projects.

**His Website**: *http://rabaix.net*

## Stefan Koopmanschap

Stefan Koopmanschap is developer, consultant and trainer. Having been involved with the symfony project as a user and advocate since late 2006, he became the Community Manager

6. http://www.apprendre-php.com
7. http://www.afup.org
8. http://www.afsy.fr
9. http://www.editions-eyrolles.com/Livre/9782212124941/symfony

for the project in the late summer of 2009. In the past, Stefan has also contributed to other open source projects including Zend Framework and phpBB.

Aside from his love for the community, Stefan has an eye for best practices and enhancing the skills of developers willing to learn more. He is a regular conference speaker and secretary of the phpBenelux[10] usergroup.

Aside from PHP, Stefan has an undying love for music and reading and above all his wife Marjolein and two kids Tomas and Yara.

**His website**: *http://www.leftontheweb.com/*

**On Twitter**: *http://twitter.com/skoop*

# Fabrice Bernhard

Fabrice Bernhard discovered the joys of web development in 1996. His first website fortunately vanished forever in the dark corners of the Internet, but the virus was caught. After studying Mathematics and Computer Science in Paris and Zürich, he became a web entrepreneur in 2007 with Allomatch.com[11], the website to find venues showing sports events in France. He then co-founded Theodo[12] in 2008, a developer-focused consulting company that specializes in quality web and open-source solutions with a strong preference for symfony.

Fabrice has been a fan of symfony and its philosophy since late 2007. He contributes to the community with the sfEasyGMap and sfFacebookConnect plugins and is looking forward to contributing further to the development of quality open-source web projects.

**His Website**: *http://www.allomatch.com/*

**His blog**: *http://www.theodo.fr/blog*

# Ryan Weaver

Ryan Weaver is lead programmer at Iostudio, LLC[13] in Nashville Tennessee and an avid supporter of symfony and other open source technologies. He is a member of the Nashville OSS group and co-organizer of the symfony user group in Nashville. Ryan loves running, reading and traveling with his girlfriend Leanna.

As a blogger, Ryan likes to share solutions and teach the methods necessary to empower other developers. He believes that a framework and its documentation should be accessible to everyone.

**His website**: *http://www.thatsquality.com/*

**On Twitter**: *http://twitter.com/weaverryan*

# Laurent Bonnet

Laurent is a web platform architect at Microsoft, focusing on both infrastructure, development and design of highly available web server farms.

10. http://www.phpbenelux.eu/
11. http://www.allomatch.com
12. http://www.theodo.fr
13. http://www.iostudio.com/

His account name at Microsoft is laurenbo, and was created in January 1992, joining as a software engineer in developer support, Windows NT Kernel. He joined the developer world in 1994 to pioneer the Regional Director program, where he recruited specialized integrators to represent Microsoft at developer seminars and conferences, and jumped into the Internet bandwagon in early 1996 as Visual Studio/Visual InterDev and Internet Explorer technical product manager. In 1998, 1999, he became the French representative of Microsoft at AFNOR (French national body of ISO), where he participated in Java standardization process. Laurent moved to an international position in 2000 to develop infrastructures for Microsoft Commercial Internet System, an eCommerce highly scalable portal solution co-developed with CompuServe, targeting Telcos and Hosters. Joining back Microsoft France in 2004, he ventured a new activity directed at hosters and web communities, and is still in that business 6 years later, helping Top hosters in France to take advantage of Windows-based solutions and products, and more recently addresses the low-cost segment which pioneered worldwide in France.

Laurent is a frequent speaker at various international events (MS-Days, Tech days, Hosting Days).

**His website**: *http://blogs.msdn.com/laurenbo*, and occasionnaly on *http://laurenbo.wordpress.com/*

**On Twitter**: *http://twitter.com/laurenbo*

Chapter 1

# Introduction

*by Fabien Potencier*

As of this writing, the symfony project has celebrated a significant milestone: its fourth birthday[14]. In just four years, the symfony framework has grown to become one of the most popular PHP frameworks in the world, powering sites such as Delicious[15], Yahoo Bookmarks[16] and Daily Motion[17]. But, with the recent release of symfony 1.4 (November 2009), we are about to end a cycle. This book is the perfect way to finish the cycle and as such, you are about to read the last book on the symfony 1 branch that will be published by the symfony project team. The next book will most likely center around symfony 2.0, to be released late 2010.

For this reason, and many others I will explain in this chapter, this book is a bit special for us.

## Why yet another book?

We have already published two books on symfony 1.3 and 1.4 recently: "Practical symfony[18]" and "The symfony reference guide[19]". The former is a great way to start learning symfony as you learn the basics of the framework through the development of a real project in a step-by-step tutorial. The latter is a reference book that holds most any symfony-related configuration information that you may need during your day-to-day development.

"More with symfony" is a book about more advanced symfony topics. This is not the first book you should read about symfony, but is one that will be helpful for people who have already developed several small projects with the framework. If you've ever wanted to know how symfony works under the hood or if you'd like to extend the framework in various ways to make it work for your specific needs, this book is for you. In this way, "More with symfony" is all about taking your symfony skills to the next level.

As this book is a collection of tutorials about various topics, feel free to read the chapters in any order, based on what you are trying to accomplish with the framework.

14. http://trac.symfony-project.org/changeset/1
15. http://sf-to.org/delicious
16. http://sf-to.org/bookmarks
17. http://sf-to.org/dailymotion
18. http://books.sensiolabs.com/book/9782918390169
19. http://books.sensiolabs.com/book/9782918390145

# About this book

This book is special because this is *a book written by the community* for the community. Dozens of people have contributed to this book: from the authors, to the translators, to the proof-readers, a large collection of effort has been put forth towards this book.

This book has been published simultaneously in no less than five languages (English, French, Italian, Spanish, and Japanese). This would not have been possible without the benevolent work of our translation teams.

This book has been made possible thanks to the *Open-Source spirit* and it is released under an Open-Source license. This fact alone changes everything. It means that nobody has been paid to work on this book: all contributors worked hard to deliver it because they were willing to do so. Each wanted to share their knowledge, give something back to the community, help spread the word about symfony and, of course, have some fun and become famous.

This book has been written by ten authors who use symfony on a day-to-day basis as developers or project managers. They have a deep knowledge of the framework and have tried to share their knowledge and experience in these chapters.

# Acknowledgments

When I started to think about writing yet another book about symfony in August 2009, I immediately had a crazy idea: what about writing a book in two months and publishing it in five languages simultaneously! Of course, involving the community in a project this big was almost mandatory. I started to talk about the idea during the PHP conference in Japan, and in a matter of hours the Japanese translation team was ready to work. That was amazing! The response from the authors and translators was equally encouraging and, in a short time, “More with symfony” was born.

I want to thank everybody who participated in one way or another during the creation of this book. This includes, in no particular order:

Ryan Weaver, Geoffrey Bachelet, Hugo Hamon, Jonathan Wage, Thomas Rabaix, Fabrice Bernhard, Kris Wallsmith, Stefan Koopmanschap, Laurent Bonnet, Julien Madelin, Franck Bodiot, Javier Eguiluz, Nicolas Ricci, Fabrizio Pucci, Francesco Fullone, Massimiliano Arione, Daniel Londero, Xavier Briand, Guillaume Bretou, Akky Akimoto, Hidenori Goto, Hideki Suzuki, Katsuhiro Ogawa, Kousuke Ebihara, Masaki Kagaya, Masao Maeda, Shin Ohno, Tomohiro Mitsumune, and Yoshihiro Takahara.

# Before we start

This book has been written for both symfony 1.3 and symfony 1.4. As writing a single book for two different versions of a software is quite unusual, this section explains what the main differences are between the two versions, and how to make the best choice for your projects.

Both the symfony 1.3 and symfony 1.4 versions have been released at about the same time (at the end of 2009). As a matter of fact, they both have the **exact same feature set**. The only difference between the two versions is how each supports backward compatibility with older symfony versions.

Symfony 1.3 is the release you’ll want to use if you need to upgrade a legacy project that uses an older symfony version (1.0, 1.1, or 1.2). It has a backward compatibility layer and all the features that have been deprecated during the 1.3 development period are still available. It means that upgrading is easy, simple, and safe.

If you start a new project today, however, you should use symfony 1.4. This version has the same feature set as symfony 1.3 but all the deprecated features, including the entire compatibility layer, have been removed. This version is cleaner and also a bit faster than symfony 1.3. Another big advantage of using symfony 1.4 is its longer support. Being a Long Term Support release, it will be maintained by the symfony core team for three years (until November 2012).

Of course, you can migrate your projects to symfony 1.3 and then slowly update your code to remove the deprecated features and eventually move to symfony 1.4 in order to benefit from the long term support. You have plenty of time to plan the move as symfony 1.3 will be supported for a year (until November 2010).

As this book does not describe deprecated features, all examples work equally well on both versions.

Chapter 2

# Advanced Routing

*by Ryan Weaver*

At its core, the routing framework is the map that links each URL to a specific location inside a symfony project and vice versa. It can easily create beautiful URLs while staying completely independent of the application logic. With advances made for in recent symfony versions, the routing framework now goes much further.

This chapter will illustrate how to create a simple web application where each client uses a separate subdomain (e.g. `client1.mydomain.com` and `client2.mydomain.com`). By extending the routing framework, this becomes quite easy.

This chapter requires that you use Doctrine as an ORM for your project.

## Project Setup: A CMS for Many Clients

In this project, an imaginary company - Sympal Builder - wants to create a CMS so that their clients can build websites as subdomains of `sympalbuilder.com`. Specifically, client XXX can view its site at `xxx.sympalbuilder.com` and use the admin area at `xxx.sympalbuilder.com/backend.php`.

The `Sympal` name was borrowed from Jonathan Wage's Sympal[20], a content management framework (CMF) built with symfony.

This project has two basic requirements:

- Users should be able to create pages and specify the title, content, and URL for those pages.
- The entire application should be built inside one symfony project that handles the frontend and backend of all client sites by determining the client and loading the correct data based on the subdomain.

To create this application, the server will need to be setup to route all `*.sympalbuilder.com` subdomains to the same document root - the web directory of the symfony project.

20. `http://www.sympalphp.org/`

## The Schema and Data

The database for the project consists of `Client` and `Page` objects. Each `Client` represents one subdomain site and consists of many `Page` objects.

*Listing 2-1*
```
# config/doctrine/schema.yml
Client:
  columns:
    name:       string(255)
    subdomain:  string(50)
  indexes:
    subdomain_index:
      fields:   [subdomain]
      type:     unique

Page:
  columns:
    title:      string(255)
    slug:       string(255)
    content:    clob
    client_id:  integer
  relations:
    Client:
      alias:        Client
      foreignAlias: Pages
      onDelete:     CASCADE
  indexes:
    slug_index:
      fields:   [slug, client_id]
      type:     unique
```

While the indexes on each table are not necessary, they are a good idea as the application will be querying frequently based on these columns.

To bring the project to life, place the following test data into the `data/fixtures/fixtures.yml` file:

*Listing 2-2*
```
# data/fixtures/fixtures.yml
Client:
  client_pete:
    name:      Pete's Pet Shop
    subdomain: pete
  client_pub:
    name:      City Pub and Grill
    subdomain: citypub

Page:
  page_pete_location_hours:
    title:     Location and Hours | Pete's Pet Shop
    content:   We're open Mon - Sat, 8 am - 7pm
    slug:      location
    Client:    client_pete
  page_pub_menu:
    title:     City Pub And Grill | Menu
    content:   Our menu consists of fish, Steak, salads, and more.
    slug:      menu
    Client:    client_pub
```

The test data introduce two websites initially, each with one page. The full URL of each page is defined by both the `subdomain` column of the `Client` and the `slug` column of the `Page` object.

Listing 2-3

```
http://pete.sympalbuilder.com/location
http://citypub.sympalbuilder.com/menu
```

### The Routing

Each page of a Sympal Builder website corresponds directly to a `Page` model object, which defines the title and content of its output. To link each URL specifically to a `Page` object, create an object route of type `sfDoctrineRoute` that uses the `slug` field. The following code will automatically look for a `Page` object in the database with a `slug` field that matches the url:

Listing 2-4

```
# apps/frontend/config/routing.yml
page_show:
  url:         /:slug
  class:       sfDoctrineRoute
  options:
    model:     Page
    type:      object
  params:
    module:    page
    action:    show
```

The above route will correctly match the `http://pete.sympalbuilder.com/location` page with the correct `Page` object. Unfortunately, the above route would *also* match the URL `http://pete.sympalbuilder.com/menu`, meaning that the restaurant's menu will be displayed on Pete's web site! At this point, the route is unaware of the importance of the client subdomains.

To bring the application to life, the route needs to be smarter. It should match the correct `Page` based on both the `slug` *and* the `client_id`, which can be determined by matching the host (e.g. `pete.sympalbuilder.com`) to the `subdomain` column on the `Client` model. To accomplish this, we'll leverage the routing framework by creating a custom routing class.

First, however, we need some background on how the routing system works.

## How the Routing System Works

A "route", in symfony, is an object of type `sfRoute` that has two important jobs:

- Generate a URL: For example, if you pass the `page_show` method a `slug` parameter, it should be able to generate a real URL (e.g. `/location`).
- Match an incoming URL: Given the URL from an incoming request, each route must be able to determine if the URL "matches" the requirements of the route.

The information for individual routes is most commonly setup inside each application's config directory located at `app/yourappname/config/routing.yml`. Recall that each route is *"an object of type* `sfRoute`*"*. So how do these simple YAML entries become `sfRoute` objects?

## Routing Cache Config Handler

Despite the fact most routes are defined in a YAML file, each entry in this file is transformed into an actual object at request time via a special type of class called a cache config handler. The final result is PHP code representing each and every route in the application. While the specifics of this process are beyond the scope of this chapter, let's peak at the final, compiled version of the `page_show` route. The compiled file is located at `cache/yourappname/envname/config/config_routing.yml.php` for the specific application and environment. Below is a shortened version of what the `page_show` route looks like:

*Listing 2-5*
```
new sfDoctrineRoute('/:slug', array (
  'module' => 'page',
  'action' => 'show',
), array (
  'slug' => '[^/\\.]+',
), array (
  'model' => 'Page',
  'type' => 'object',
));
```

The class name of each route is defined by the `class` key inside the `routing.yml` file. If no `class` key is specified, the route will default to be a class of `sfRoute`. Another common route class is `sfRequestRoute` which allows the developer to create RESTful routes. A full list of route classes and available options is available via The symfony Reference Book[21]

## Matching an Incoming Request to a Specific Route

One of the main jobs of the routing framework is to match each incoming URL with the correct route object. The `sfPatternRouting` class represents the core routing engine and is tasked with this exact task. Despite its importance, a developer will rarely interact directly with `sfPatternRouting`.

To match the correct route, `sfPatternRouting` iterates through each `sfRoute` and "asks" the route if it matches the incoming url. Internally, this means that `sfPatternRouting` calls the `sfRoute::matchesUrl()` method on each route object. This method simply returns `false` if the route doesn't match the incoming url.

However, if the route *does* match the incoming URL, `sfRoute::matchesUrl()` does more than simply return `true`. Instead, the route returns an array of parameters that are merged into the request object. For example, the url `http://pete.sympalbuilder.com/location` matches the `page_show` route, whose `matchesUrl()` method would return the following array:

*Listing 2-6*
```
array('slug' => 'location')
```

This information is then merged into the request object, which is why it's possible to access route variables (e.g. `slug`) from the actions file and other places.

*Listing 2-7*
```
$this->slug = $request->getParameter('slug');
```

As you may have guessed, overriding the `sfRoute::matchesUrl()` method is a great way to extend and customize a route to do almost anything.

21. `http://www.symfony-project.org/reference/1_3/en/10-Routing`

# Creating a Custom Route Class

In order to extend the `page_show` route to match based on the subdomain of the `Client` objects, we will create a new custom route class. Create a file named `acClientObjectRoute.class.php` and place it in the project's `lib/routing` directory (you'll need to create this directory):

*Listing 2-8*

```
// lib/routing/acClientObjectRoute.class.php
class acClientObjectRoute extends sfDoctrineRoute
{
  public function matchesUrl($url, $context = array())
  {
    if (false === $parameters = parent::matchesUrl($url, $context))
    {
      return false;
    }

    return $parameters;
  }
}
```

The only other step is to instruct the `page_show` route to use this route class. In `routing.yml`, update the `class` key on the route:

*Listing 2-9*

```
# apps/fo/config/routing.yml
page_show:
  url:        /:slug
  class:      acClientObjectRoute
  options:
    model:    Page
    type:     object
  params:
    module:   page
    action:   show
```

So far, `acClientObjectRoute` adds no additional functionality, but all the pieces are in place. The `matchesUrl()` method has two specific jobs.

## Adding Logic to the Custom Route

To give the custom route the needed functionality, replace the contents of the `acClientObjectRoute.class.php` file with the following.

*Listing 2-10*

```
class acClientObjectRoute extends sfDoctrineRoute
{
  protected $baseHost = '.sympalbuilder.com';

  public function matchesUrl($url, $context = array())
  {
    if (false === $parameters = parent::matchesUrl($url, $context))
    {
      return false;
    }

    // return false if the baseHost isn't found
    if (strpos($context['host'], $this->baseHost) === false)
```

```
    {
      return false;
    }

    $subdomain = str_replace($this->baseHost, '', $context['host']);

    $client = Doctrine_Core::getTable('Client')
      ->findOneBySubdomain($subdomain)
    ;

    if (!$client)
    {
      return false;
    }

    return array_merge(array('client_id' => $client->id), $parameters);
  }
}
```

The initial call to `parent::matchesUrl()` is important as it runs through the normal route-matching process. In this example, since the URL `/location` matches the `page_show` route, `parent::matchesUrl()` would return an array containing the matched `slug` parameter.

*Listing 2-11*
```
array('slug' => 'location')
```

In other words, all the hard-work of route matching is done for us, which allows the remainder of the method to focus on matching based on the correct `Client` subdomain.

*Listing 2-12*
```
public function matchesUrl($url, $context = array())
{
  // ...

  $subdomain = str_replace($this->baseHost, '', $context['host']);

  $client = Doctrine_Core::getTable('Client')
    ->findOneBySubdomain($subdomain)
  ;

  if (!$client)
  {
    return false;
  }

  return array_merge(array('client_id' => $client->id), $parameters);
}
```

By performing a simple string replace, we can isolate the subdomain portion of the host and then query the database to see if any of the `Client` objects have this subdomain. If no `Client` objects match the subdomain, then we return `false` indicating that the incoming request does not match the route. However, if there is a `Client` object with the current subdomain, we merge an extra parameter, `client_id` into the returned array.

The `$context` array passed to `matchesUrl()` is prepopulated with lot's of useful information about the current request, including the `host`, an `is_secure` boolean, the `request_uri`, the HTTP `method` and more.

But, what has the custom route really accomplished? The `acClientObjectRoute` class now does the following:

- The incoming `$url` will only match if the `host` contains a subdomain belonging to one of the `Client` objects.
- If the route matches, an additional `client_id` parameter for the matched `Client` object is returned and ultimately merged into the request parameters.

## Leveraging the Custom Route

Now that the correct `client_id` parameter is being returned by `acClientObjectRoute`, we have access to it via the request object. For example, the `page/show` action could use the `client_id` to find the correct `Page` object:

*Listing 2-13*

```
public function executeShow(sfWebRequest $request)
{
  $this->page = Doctrine_Core::getTable('Page')->findOneBySlugAndClientId(
    $request->getParameter('slug'),
    $request->getParameter('client_id')
  );

  $this->forward404Unless($this->page);
}
```

The `findOneBySlugAndClientId()` method is a type of magic finder[22] new in Doctrine 1.2 that queries for objects based on multiple fields.

As nice as this is, the routing framework allows for an even more elegant solution. First, add the following method to the `acClientObjectRoute` class:

*Listing 2-14*

```
protected function getRealVariables()
{
  return array_merge(array('client_id'), parent::getRealVariables());
}
```

With this final piece, the action can rely completely on the route to return the correct `Page` object. The `page/show` action can be reduced to a single line.

*Listing 2-15*

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

Without any additional work, the above code will query for a `Page` object based on both the `slug` *and* `client_id` columns. Additionally, like all object routes, the action will automatically forward to a 404 page if no corresponding object is found.

But how does this work? Object routes, like `sfDoctrineRoute`, which the `acClientObjectRoute` class extends, automatically query for the related object based on the variables in the `url` key of the route. For example, the `page_show` route, which contains the `:slug` variable in its `url`, queries for the `Page` object via the `slug` column.

In this application, however, the `page_show` route must also query for `Page` objects based on the `client_id` column. To do this, we've overridden the

22. `http://www.doctrine-project.org/upgrade/1_2#Expanded%20Magic%20Finders%20to%20Multiple%20Fields`

`sfObjectRoute::getRealVariables()`, which is called internally to determine which columns to use for the object query. By adding the `client_id` field to this array, the `acClientObjectRoute` will query based on both the `slug` and `client_id` columns.

Objects routes automatically ignore any variables that don't correspond to a real column. For example, if the URL key contains a `:page` variable, but no `page` column exists on the relevant table, the variable will be ignored.

At this point, the custom route class accomplishes everything needed with very little effort. In the next sections, we'll reuse the new route to create a client-specific admin area.

### Generating the Correct Route

One small problem remains with how the route is generated. Suppose create a link to a page with the following code:

*Listing 2-16*
```
<?php echo link_to('Locations', 'page_show', $page) ?>
```

*Listing 2-17*
```
Generated url: /location?client_id=1
```

As you can see, the `client_id` was automatically appended to the url. This occurs because the route tries to use all its available variables to generate the url. Since the route is aware of both a `slug` parameter and a `client_id` parameter, it uses both when generating the route.

To fix this, add the following method to the `acClientObjectRoute` class:

*Listing 2-18*
```
protected function doConvertObjectToArray($object)
{
  $parameters = parent::doConvertObjectToArray($object);

  unset($parameters['client_id']);

  return $parameters;
}
```

When an object route is generated, it attempts to retrieve all of the necessary information by calling `doConvertObjectToArray()`. By default, the `client_id` is returned in the `$parameters` array. By unsetting it, however, we prevent it from being included in the generated url. Remember that we have this luxury since the `Client` information is held in the subdomain itself.

You can override the `doConvertObjectToArray()` process entirely and handle it yourself by adding a `toParams()` method to the model class. This method should return an array of the parameters that you want to be used during route generation.

## Route Collections

To finish the Sympal Builder application, we need to create an admin area where each individual `Client` can manage its `Pages`. To do this, we will need a set of actions that allows us to list, create, update, and delete the `Page` objects. As these types of modules are fairly common, symfony can generate the module automatically. Execute the following task from the command line to generate a `pageAdmin` module inside an application called `backend`:

*Listing 2-19*

```
$ php symfony doctrine:generate-module backend pageAdmin Page
--with-doctrine-route --with-show
```

The above task generates a module with an actions file and related templates capable of making all the modifications necessary to any `Page` object. Lot's of customizations could be made to this generated CRUD, but that falls outside the scope of this chapter.

While the above task prepares the module for us, we still need to create a route for each action. By passing the `--with-doctrine-route` option to the task, each action was generated to work with an object route. This decreases the amount of code in each action. For example, the `edit` action contains one simple line:

*Listing 2-20*

```
public function executeEdit(sfWebRequest $request)
{
  $this->form = new PageForm($this->getRoute()->getObject());
}
```

In total, we need routes for the `index`, `new`, `create`, `edit`, `update`, and `delete` actions. Normally, creating these routes in a RESTful[23] manner would require significant setup in `routing.yml`.

*Listing 2-21*

```
pageAdmin:
  url:          /pages
  class:        sfDoctrineRoute
  options:      { model: Page, type: list }
  params:       { module: page, action: index }
  requirements:
    sf_method: [get]
pageAdmin_new:
  url:         /pages/new
  class:       sfDoctrineRoute
  options:     { model: Page, type: object }
  params:      { module: page, action: new }
  requirements:
    sf_method: [get]
pageAdmin_create:
  url:         /pages
  class:       sfDoctrineRoute
  options:     { model: Page, type: object }
  params:      { module: page, action: create }
  requirements:
    sf_method: [post]
pageAdmin_edit:
  url:         /pages/:id/edit
  class:       sfDoctrineRoute
  options:     { model: Page, type: object }
  params:      { module: page, action: edit }
  requirements:
    sf_method: [get]
pageAdmin_update:
  url:         /pages/:id
  class:       sfDoctrineRoute
  options:     { model: Page, type: object }
  params:      { module: page, action: update }
  requirements:
    sf_method: [put]
```

23. http://en.wikipedia.org/wiki/Representational State Transfer

```
pageAdmin_delete:
  url:        /pages/:id
  class:      sfDoctrineRoute
  options:    { model: Page, type: object }
  params:     { module: page, action: delete }
  requirements:
    sf_method: [delete]
pageAdmin_show:
  url:        /pages/:id
  class:      sfDoctrineRoute
  options:    { model: Page, type: object }
  params:     { module: page, action: show }
  requirements:
    sf_method: [get]
```

To visualize these routes, use the `app:routes` task, which displays a summary of every route for a specific application:

*Listing 2-22*
```
$ php symfony app:routes backend

>> app       Current routes for application "backend"
Name             Method Pattern
pageAdmin        GET    /pages
pageAdmin_new    GET    /pages/new
pageAdmin_create POST   /pages
pageAdmin_edit   GET    /pages/:id/edit
pageAdmin_update PUT    /pages/:id
pageAdmin_delete DELETE /pages/:id
pageAdmin_show   GET    /pages/:id
```

## Replacing the Routes with a Route Collection

Fortunately, symfony provides a much easier way to specify all of the routes that belong to a traditional CRUD. Replace the entire content of `routing.yml` with one simple route.

*Listing 2-23*
```
pageAdmin:
  class:   sfDoctrineRouteCollection
  options:
    model:        Page
    prefix_path:  /pages
    module:       pageAdmin
```

Once again, execute the `app:routes` task to visualize all of the routes. As you'll see, all seven of the previous routes still exist.

*Listing 2-24*
```
$ php symfony app:routes backend

>> app       Current routes for application "backend"
Name             Method Pattern
pageAdmin        GET    /pages.:sf_format
pageAdmin_new    GET    /pages/new.:sf_format
pageAdmin_create POST   /pages.:sf_format
pageAdmin_edit   GET    /pages/:id/edit.:sf_format
pageAdmin_update PUT    /pages/:id.:sf_format
pageAdmin_delete DELETE /pages/:id.:sf_format
pageAdmin_show   GET    /pages/:id.:sf_format
```

Route collections are a special type of route object that internally represent more than one route. The `sfDoctrineRouteCollection` route, for example automatically generates the seven most common routes needed for a CRUD. Behind the scenes, `sfDoctrineRouteCollection` is doing nothing more than creating the same seven routes previously specified in `routing.yml`. Route collections basically exist as a shortcut to creating a common group of routes.

# Creating a Custom Route Collection

At this point, each `Client` will be able to modify its `Page` objects inside a functioning crud via the URL `/pages`. Unfortunately, each `Client` can currently see and modify *all* `Page` objects - those both belonging and not belonging to the `Client`. For example, `http://pete.sympalbuilder.com/backend.php/pages` will render a list of *both* pages in the fixtures - the `location` page from Pete's Pet Shop and the `menu` page from City Pub.

To fix this, we'll reuse the `acClientObjectRoute` that was created for the frontend. The `sfDoctrineRouteCollection` class generates a group of `sfDoctrineRoute` objects. In this application, we'll need to generate a group of `acClientObjectRoute` objects instead.

To accomplish this, we'll need to use a custom route collection class. Create a new file named `acClientObjectRouteCollection.class.php` and place it in the `lib/routing` directory. Its content is incredibly straightforward:

*Listing 2-25*

```
// lib/routing/acClientObjectRouteCollection.class.php
class acClientObjectRouteCollection extends sfObjectRouteCollection
{
  protected
    $routeClass = 'acClientObjectRoute';
}
```

The `$routeClass` property defines the class that will be used when creating each underlying route. Now that each underlying routing is an `acClientObjectRoute` route, the job is actually done. For example, `http://pete.sympalbuilder.com/backend.php/pages` will now list only *one* page: the `location` page from Pete's Pet Shop. Thanks to the custom route class, the index action returns only `Page` objects related to the correct `Client`, based on the subdomain of the request. With just a few lines of code, we've created an entire backend module that can be safely used by multiple clients.

## Missing Piece: Creating New Pages

Currently, a `Client` select box displays on the backend when creating or editing `Page` objects. Instead of allowing users to choose the `Client` (which would be a security risk), let's set the `Client` automatically based on the current subdomain of the request.

First, update the `PageForm` object in `lib/form/PageForm.class.php`.

*Listing 2-26*

```
public function configure()
{
  $this->useFields(array(
    'title',
    'content',
  ));
}
```

The select box is now missing from the `Page` forms as needed. However, when new `Page` objects are created, the `client_id` is never set. To fix this, manually set the related `Client` in both the `new` and `create` actions.

Listing 2-27
```
public function executeNew(sfWebRequest $request)
{
  $page = new Page();
  $page->Client = $this->getRoute()->getClient();
  $this->form = new PageForm($page);
}
```

This introduces a new function, `getClient()` which doesn't currently exist in the `acClientObjectRoute` class. Let's add it to the class by making a few simple modifications:

Listing 2-28
```
// lib/routing/acClientObjectRoute.class.php
class acClientObjectRoute extends sfDoctrineRoute
{
  // ...

  protected $client = null;

  public function matchesUrl($url, $context = array())
  {
    // ...

    $this->client = $client;

    return array_merge(array('client_id' => $client->id), $parameters);
  }

  public function getClient()
  {
    return $this->client;
  }
}
```

By adding a `$client` class property and setting it in the `matchesUrl()` function, we can easily make the `Client` object available via the route. The `client_id` column of new `Page` objects will now be automatically and correctly set based on the subdomain of the current host.

# Customizing an Object Route Collection

By using the routing framework, we have now easily solved the problems posed by creating the Sympal Builder application. As the application grows, the developer will be able to reuse the custom routes for other modules in the backend area (e.g., so each `Client` can manage their photo galleries).

Another common reason to create a custom route collection is to add additional, commonly used routes. For example, suppose a project employs many models, each with an `is_active` column. In the admin area, there needs to be an easy way to toggle the `is_active` value for any particular object. First, modify `acClientObjectRouteCollection` and instruct it to add a new route to the collection:

Listing 2-29
```
// lib/routing/acClientObjectRouteCollection.class.php
protected function generateRoutes()
```

```
{
  parent::generateRoutes();

  if (isset($this->options['with_is_active']) &&
$this->options['with_is_active'])
  {
    $routeName = $this->options['name'].'_toggleActive';

    $this->routes[$routeName] = $this->getRouteForToggleActive();
  }
}
```

The `sfObjectRouteCollection::generateRoutes()` method is called when the collection object is instantiated and is responsible for creating all the needed routes and adding them to the `$routes` class property array. In this case, we offload the actual creation of the route to a new protected method called `getRouteForToggleActive()`:

*Listing 2-30*

```
protected function getRouteForToggleActive()
{
  $url = sprintf(
    '%s/:%s/toggleActive.:sf_format',
    $this->options['prefix_path'],
    $this->options['column']
  );

  $params = array(
    'module' => $this->options['module'],
    'action' => 'toggleActive',
    'sf_format' => 'html'
  );

  $requirements = array('sf_method' => 'put');

  $options = array(
    'model' => $this->options['model'],
    'type' => 'object',
    'method' => $this->options['model_methods']['object']
  );

  return new $this->routeClass(
    $url,
    $params,
    $requirements,
    $options
  );
}
```

The only remaining step is to setup the route collection in `routing.yml`. Notice that `generateRoutes()` looks for an option named `with_is_active` before adding the new route. Adding this logic gives us more control in case we want to use the `acClientObjectRouteCollection` somewhere later that doesn't need the `toggleActive` route:

*Listing 2-31*

```
# apps/frontend/config/routing.yml
pageAdmin:
  class:   acClientObjectRouteCollection
  options:
    model:          Page
```

```
    prefix_path:    /pages
    module:         pageAdmin
    with_is_active: true
```

Check the `app:routes` task and verify that the new `toggleActive` route is present. The only remaining piece is to create the action that will do that actual work. Since you may want to use this route collection and corresponding action across several modules, create a new `backendActions.class.php` file in the `apps/backend/lib/action` directory (you'll need to create this directory):

*Listing 2-32*

```
# apps/backend/lib/action/backendActions.class.php
class backendActions extends sfActions
{
  public function executeToggleActive(sfWebRequest $request)
  {
    $obj = $this->getRoute()->getObject();

    $obj->is_active = !$obj->is_active;

    $obj->save();

    $this->redirect($this->getModuleName().'/index');
  }
}
```

Finally, change the base class of the `pageAdminActions` class to extend this new `backendActions` class.

*Listing 2-33*

```
class pageAdminActions extends backendActions
{
  // ...
}
```

What have we just accomplished? By adding a route to the route collection and an associated action in a base actions file, any new module can automatically use this functionality simply by using the `acClientObjectRouteCollection` and extending the `backendActions` class. In this way, common functionality can be easily shared across many modules.

# Options on a Route Collection

Object route collections contain a series of options that allow it to be highly customized. In many cases, a developer can use these options to configure the collection without needing to create a new custom route collection class. A detailed list of route collection options is available via The symfony Reference Book[24].

## Action Routes

Each object route collection accepts three different options which determine the exact routes generated in the collection. Without going into great detail, the following collection would generate all seven of the default routes along with an additional collection route and object route:

24. `http://www.symfony-project.org/reference/1_3/en/10-Routing#chapter_10_sfobjectroutecollection`

Listing 2-34
```
pageAdmin:
  class:   acClientObjectRouteCollection
  options:
    # ...
    actions:      [list, new, create, edit, update, delete, show]
    collection_actions:
      indexAlt:   [get]
    object_actions:
      toggle:     [put]
```

### Column

By default, the primary key of the model is used in all of the generated urls and is used to query for the objects. This, of course, can easily be changed. For example, the following code would use the `slug` column instead of the primary key:

Listing 2-35
```
pageAdmin:
  class:   acClientObjectRouteCollection
  options:
    # ...
    column: slug
```

### Model Methods

By default, the route retrieves all related objects for a collection route and queries on the specified `column` for object routes. If you need to override this, add the `model_methods` option to the route. In this example, the `fetchAll()` and `findForRoute()` methods would need to be added to the `PageTable` class. Both methods will receive an array of request parameters as an argument:

Listing 2-36
```
pageAdmin:
  class:   acClientObjectRouteCollection
  options:
    # ...
    model_methods:
      list:       fetchAll
      object:     findForRoute
```

### Default Parameters

Finally, suppose that you need to make a specific request parameter available in the request for each route in the collection. This is easily done with the `default_params` option:

Listing 2-37
```
pageAdmin:
  class:   acClientObjectRouteCollection
  options:
    # ...
    default_params:
      foo:   bar
```

## Final Thoughts

The traditional job of the routing framework - to match and generate urls - has evolved into a fully customizable system capable of catering to the most complex URL requirements of a project. By taking control of the route objects, the special URL structure can be abstracted

away from the business logic and kept entirely inside the route where it belongs. The end result is more control, more flexibility and more manageable code.

Chapter 3

# Enhance your Productivity

*by Fabien Potencier*

Using symfony itself is a great way to enhance your productivity as a web developer. Of course, everyone already knows how symfony's detailed exceptions and web debug toolbar can greatly enhance productivity. This chapter will teach you some tips and tricks to enhance your productivity even more by using some new or less well-known symfony features.

## Start Faster: Customize the Project Creation Process

Thanks to the symfony CLI tool, creating a new symfony project is quick and simple:

```
$ php /path/to/symfony generate:project foo --orm=Doctrine
```

*Listing 3-1*

The `generate:project` task generates the default directory structure for your new project and creates configuration files with sensible defaults. You can then use other symfony tasks to create applications, install plugins, configure your model, and more.

But the first steps to create a new project are usually always quite the same: you create a main application, install a bunch of plugins, tweak some configuration defaults to your liking, and so on.

As of symfony 1.3, the project creation process can be customized and automated.

As all symfony tasks are classes, it's pretty easy to customize and extend them except. The `generate:project` task, however, cannot be easily customized because no project exists when the task is executed.

The `generate:project` task takes an `--installer` option, which is a PHP script that will be executed during the project creation process:

```
$ php /path/to/symfony generate:project --installer=/somewhere/
my_installer.php
```

*Listing 3-2*

The `/somewhere/my_installer.php` script will be executed in the context of the `sfGenerateProjectTask` instance, so it has access to the task's methods to by using the `$this` object. The following sections describe all the available methods you can use to customize your project creation process.

If you enable URL file-access for the `include()` function in your `php.ini`, you can even pass a URL as an installer (of course you need to be very careful when doing this with a script you know nothing about):

Listing 3-3
```
$ symfony generate:project
--installer=http://example.com/sf_installer.php
```

### installDir()

The `installDir()` method mirrors a directory structure (composed of sub-directories and files) in the newly created project:

Listing 3-4
```
$this->installDir(dirname(__FILE__).'/skeleton');
```

### runTask()

The `runTask()` method executes a task. It takes the task name, and a string representing the arguments and the options you want to pass to it as arguments:

Listing 3-5
```
$this->runTask('configure:author', "'Fabien Potencier'");
```

Arguments and options can also be passed as arrays:

Listing 3-6
```
$this->runTask('configure:author', array('author' => 'Fabien Potencier'));
```

The task shortcut names also work as expected:

Listing 3-7
```
$this->runTask('cc');
```

This method can of course be used to install plugins:

Listing 3-8
```
$this->runTask('plugin:install', 'sfDoctrineGuardPlugin');
```

To install a specific version of a plugin, just pass the needed options:

Listing 3-9
```
$this->runTask('plugin:install', 'sfDoctrineGuardPlugin', array('release'
=> '10.0.0', 'stability' => beta'));
```

To execute a task from a freshly installed plugin, the tasks need to be reloaded first:

Listing 3-10
```
$this->reloadTasks();
```

If you create a new application and want to use tasks that relys on a specific application like `generate:module`, you must change the configuration context yourself:

Listing 3-11
```
$this->setConfiguration($this->createConfiguration('frontend', 'dev'));
```

## Loggers

To give feedback to the developer when the installer script runs, you can log things pretty easily:

Listing 3-12
```
// a simple log
$this->log('some installation message');
```

```
// log a block
$this->logBlock('Fabien\'s Crazy Installer', 'ERROR_LARGE');

// log in a section
$this->logSection('install', 'install some crazy files');
```

## User Interaction

The `askConfirmation()`, `askAndValidate()`, and `ask()` methods allow you to ask questions and make your installation process dynamically configurable.

If you just need a confirmation, use the `askConfirmation()` method:

*Listing 3-13*

```
if (!$this->askConfirmation('Are you sure you want to run this crazy
installer?'))
{
  $this->logSection('install', 'You made the right choice!');

  return;
}
```

You can also ask any question and get the user's answer as a string by using the `ask()` method:

*Listing 3-14*

```
$secret = $this->ask('Give a unique string for the CSRF secret:');
```

And if you want to validate the answer, use the `askAndValidate()` method:

*Listing 3-15*

```
$validator = new sfValidatorEmail(array(), array('invalid' => 'hmmm, it
does not look like an email!'));
$email = $this->askAndValidate('Please, give me your email:', $validator);
```

## Filesystem Operations

If you want to do filesystem changes, you can access the symfony filesystem object:

*Listing 3-16*

```
$this->getFilesystem()->...();
```

### The Sandbox Creation Process

The symfony sandbox is a pre-packaged symfony project with a ready-made application and a pre-configured SQLite database. Anybody can create a sandbox by using its installer script:

*Listing 3-17*

```
$ php symfony generate:project --installer=/path/to/symfony/data/bin/
sandbox_installer.php
```

Have a look at the `symfony/data/bin/sandbox_installer.php` script to have a working example of an installer script.

The installer script is just another PHP file. So, you can do pretty anything you want. Instead of running the same tasks again and again each time you create a new symfony project, you can create your own installer script and tweak your symfony project installations the way you want. Creating a new project with an installer is much faster and prevents you from missing steps. You can even share your installer script with others!

In Chapter 06 *(page 66)*, we will use a custom installer. The code for it can be found in Appendix B *(page 239)*.

# Develop Faster

From PHP code to CLI tasks, programming means a lot of typing. Let's see how to reduce this to the bare minimum.

## Choosing your IDE

Using an IDE helps the developer to be more productive in more than one way.

First, most modern IDEs provide PHP autocompletion out of the box. This means that you only need to type the first few character of a method name. This also means that even if you don't remember the method name, you are not forced to have look at the API as the IDE will suggest all the available methods of the current object.

Additionally, some IDEs, like PHPEdit or Netbeans, know even more about symfony and provide specific integration with symfony projects.

**Text Editors**

Some users prefer to use a text editor for their programming work, mainly because text editors are faster than any IDE. Of course, text editors provide less IDE-oriented features. Most popular editors, however, offer plugins/extensions that can be used to enhance your user experience and make the editor work more efficiently with PHP and symfony projects.

For example, a lot of Linux users tends to use VIM for all their work. For these developers, the vim-symfony[25] extension is available. VIM-symfony is a set of VIM scripts that integrates symfony into your favorite editor. Using vim-symfony, you can easily create vim macros and commands to streamline your symfony development. It also bundles a set of default commands that put a number of configuration files at your fingertips (schema, routing, etc) and enable you to easily switch from actions to templates.

Some MacOS X users use TextMate. These developers can install the symfony bundle[26], which adds a lot of time-saving macros and shortcuts for day-to-day activities.

### Using an IDE that supports symfony

Some IDEs, like PHPEdit 3.4[27] and NetBeans 6.8[28], have native support for symfony, and so provide a finely-grained integration with the framework. Have a look at their documentation to learn more about their symfony specific support, and how it can help you develop faster.

### Helping the IDE

PHP autocompletion in IDEs only works for methods that are explicitly defined in the PHP code. But if your code uses the `__call()` or `__get()` "magic" methods, IDEs have no way to guess the available methods or properties. The good news is that you can help most IDEs by providing the methods and/or properties in a PHPDoc block (by using the `@method` and `@property` annotations respectively).

25. `http://github.com/geoffrey/vim-symfony`
26. `http://github.com/denderello/symfony-tmbundle`
27. `http://www.phpedit.com/en/presentation/extensions/symfony`
28. `http://www.netbeans.org/community/releases/68/`

Let's say you have a `Message` class with a dynamic property (`message`) and a dynamic method (`getMessage()`). The following code shows you how an IDE can know about them without any explicit definition in the PHP code:

Listing 3-18

```
/**
 * @property clob $message
 *
 * @method clob getMessage() Returns the current message value
 */
class Message
{
  public function __get()
  {
    // ...
  }

  public function __call()
  {
    // ...
  }
}
```

Even if the `getMessage()` method does not exist, it will be recognized by the IDE thanks to the `@method` annotation. The same goes for the `message` property as we have added a `@property` annotation.

This technique is used by the `doctrine:build-model` task. For instance, a Doctrine `MailMessage` class with two columns (`message` and `priority`) looks like the following:

Listing 3-19

```
/**
 * BaseMailMessage
 *
 * This class has been auto-generated by the Doctrine ORM Framework
 *
 * @property clob $message
 * @property integer $priority
 *
 * @method clob        getMessage()  Returns the current record's
"message" value
 * @method integer     getPriority() Returns the current record's
"priority" value
 * @method MailMessage setMessage()  Sets the current record's "message"
value
 * @method MailMessage setPriority() Sets the current record's "priority"
value
 *
 * @package    ##PACKAGE##
 * @subpackage ##SUBPACKAGE##
 * @author     ##NAME## <##EMAIL##>
 * @version    SVN: $Id: Builder.php 6508 2009-10-14 06:28:49Z jwage $
 */
abstract class BaseMailMessage extends sfDoctrineRecord
{
    public function setTableDefinition()
    {
        $this->setTableName('mail_message');
        $this->hasColumn('message', 'clob', null, array(
             'type' => 'clob',
```

```
            'notnull' => true,
            ));
        $this->hasColumn('priority', 'integer', null, array(
            'type' => 'integer',
            ));
    }

    public function setUp()
    {
        parent::setUp();
        $timestampable0 = new Doctrine_Template_Timestampable();
        $this->actAs($timestampable0);
    }
}
```

# Find Documentation Faster

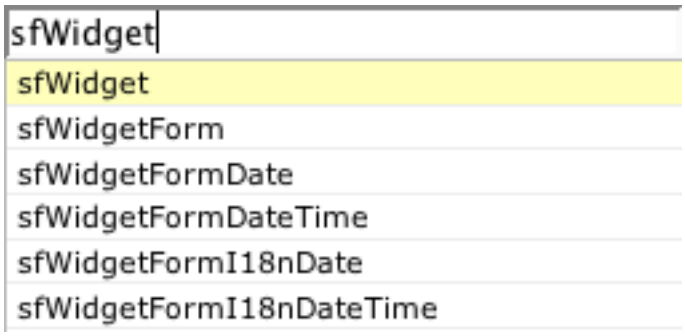
As symfony is a large framework with many features, it is not always easy to remember all the configuration possibilities, or all the classes and methods at your disposal. As we have seen before, using an IDE can go a long way in providing you autocompletion. Let's explore how existing tools can be leveraged to find answers as fast as possible.

## Online API

The fastest way to find documentation about a class or a method is to browse the online API[29].

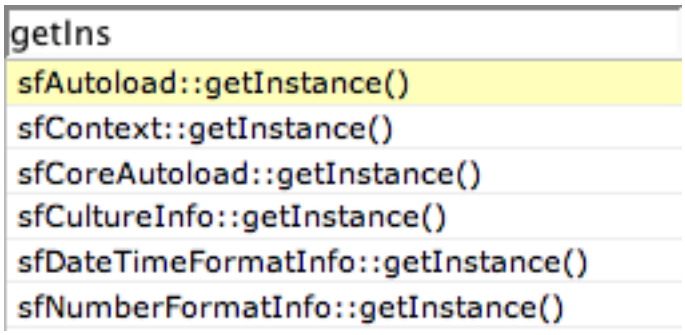
Even more interesting is the built-in API search engine. The search allows you to rapidly find a class or a method with only a few keystrokes. After entering a few letters into the search box of the API page, a quick search results box will appear in real-time with useful suggestions.

You can search by typing the beginning of a class name:

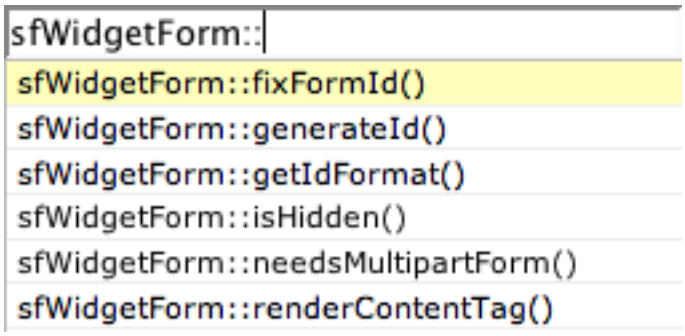



or of a method name:

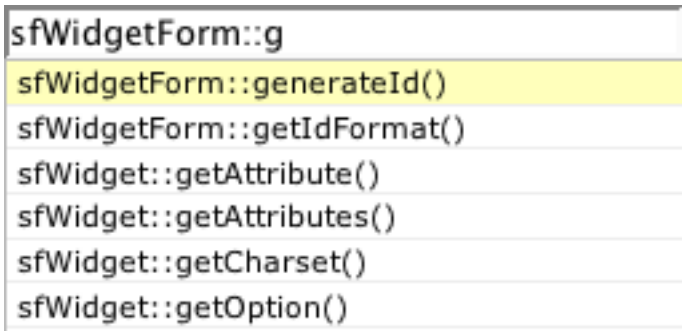
29. http://www.symfony-project.org/api/1_3/




or a class name followed by `::` to list all available methods:




or enter the beginning of a method name to further refine the possibilities:




If you want to list all classes of a package, just type the package name and submit the request.

You can even integrate the symfony API search in your browser. This way, you don't even need to navigate to the symfony website to look for something. This is possible because we provide native OpenSearch[30] support for the symfony API.

If you use Firefox, the symfony API search engines will show up automatically in the search engine menu. You can also click on the "API OpenSearch" link from the API documentation section to add one of them to your browser search box.

You can have a look at a screencast that shows how the symfony API search engine integrates well with Firefox on the symfony blog[31].

## Cheat Sheets

If you want to quickly access information about the main parts of the framework, a large collection of cheat sheets[32] is available:

30. `http://www.opensearch.org/`
31. `http://www.symfony-project.org/blog/2009/02/24/opensearch-support-for-the-symfony-api`
32. `http://trac.symfony-project.org/wiki/CheatSheets`

- Directory Structure and CLI[33]
- View[34]
- View: Partials, Components, Slots and Component Slots[35]
- Lime Unit & Functional Testing[36]
- ORM[37]
- Propel[38]
- Propel Schema[39]
- Doctrine[40]

Some of these cheat sheets have not yet been updated for symfony 1.3.

### Offline Documentation

Questions about configuration are best answered by the symfony reference guide. This is a book you should keep with you whenever you develop with symfony. The book is the fastest way to find every available configuration thanks to a very detailed table of contents, an index of terms, cross-references inside the chapters, tables, and much more.

You can browse this book online[41], buy a printed[42] copy of it, or even download a PDF[43] version.

### Online Tools

As seen at the beginning of this chapter, symfony provides a nice toolset to help you get started faster. Eventually, you will finish your project, and it will be time to deploy it to production.

To check that your project is ready for deployment, you can use the online deployment checklist[44]. This website covers the major points you need to check before going to production.

## Debug Faster

When an error occurs in the development environment, symfony displays a nice exception page filled with useful information. You can, for instance, have a look at the stack trace and

33. http://andreiabohner.files.wordpress.com/2007/03/cheatsheetsymfony001_enus.pdf
34. http://andreiabohner.files.wordpress.com/2007/08/sfviewfirstpartrefcard.pdf
35. http://andreiabohner.files.wordpress.com/2007/08/sfviewsecondpartrefcard.pdf
36. http://trac.symfony-project.com/attachment/wiki/LimeTestingFramework/lime-cheat.pdf?format=raw
37. http://andreiabohner.files.wordpress.com/2007/08/sform_enus.pdf
38. http://andreiabohner.files.wordpress.com/2007/08/sfmodelfirstpartrefcard.pdf
39. http://andreiabohner.files.wordpress.com/2007/09/sfmodelsecondpartrefcard.pdf
40. http://www.phpdoctrine.org/Doctrine-Cheat-Sheet.pdf
41. http://www.symfony-project.org/reference/1_3/en/
42. http://books.sensiolabs.com/book/the-symfony-1-3-reference-guide
43. http://www.symfony-project.org/get/pdf/reference-1.3-en.pdf
44. http://symfony-check.org/

the files that have been executed. If you setup the `sf_file_link_format` setting in the `settings.yml` configuration file (see below), you can even click on the filenames and the related file will be opened at the right line in your favorite text editor or IDE. This is a great example of a very small feature that can save you tons of time when debugging a problem.

The log and view panels in the web debug toolbar also display filenames (especially when XDebug is enabled) that become clickable when you set the `sf_file_link_format` setting.

By default, the `sf_file_link_format` is empty and symfony defaults to the value of the `xdebug.file_link_format`[45] PHP configuration value if it exists (setting `xdebug.file_link_format` in `php.ini` allows recent versions of XDebug to add links for all filenames in the stack trace).

The value for `sf_file_link_format` depends on your IDE and Operating System. For instance, if you want to open files in TextMate, add the following to `settings.yml`:

*Listing 3-20*

```
dev:
  .settings:
    file_link_format: txmt://open?url=file://%f&line=%l
```

The `%f` placeholder is replaced by symfony with the absolute path of the file and the `%l` placeholder is replaced with the line number.

If you use VIM, the configuration is more involved and is described online for symfony[46] and XDebug[47].

Use your favorite search engine to learn how to configure your IDE. You can look for configuration of the `sf_file_link_format` or `xdebug.file_link_format` as both work in the same way.

# Test Faster

## Record Your Functional Tests

Functional tests simulate user interaction to thoroughly test the integration of all the pieces of your application. Writing functional tests is easy but time consuming. But as each functional test file is a scenario that simulates a user browsing your website, and because browsing an application is faster than writing PHP code, what if you could record a browser session and have it automatically converted to PHP code? Thankfully, symfony has such a plugin. It's called swFunctionalTestGenerationPlugin[48], and it allows you to generate ready-to-be-customized test skeletons in a matter of minutes. Of course, you will still need to add the proper tester calls to make it useful, but this is nonetheless a great time-saver.

The plugin works by registering a symfony filter that will intercept all requests, and convert them to functional test code. After installing the plugin the usual way, you need to enable it.

45. `http://xdebug.org/docs/all_settings#file_link_format`
46. `http://geekblog.over-blog.com/article-symfony-open-exceptions-files-in-remote-vim-sessions-37895120.html`
47. `http://www.koch.ro/blog/index.php?/archives/77-Firefox,-VIM,-Xdebug-Jumping-to-the-error-line.html`
48. `http://www.symfony-project.org/plugins/swFunctionalTestGenerationPlugin`

Open the `filters.yml` of your application and add the following lines after the comment line:

Listing 3-21
```
functional_test:
  class: swFilterFunctionalTest
```

Next, enable the plugin in your `ProjectConfiguration` class:

Listing 3-22
```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    // ...

    $this->enablePlugin('swFunctionalTestGenerationPlugin');
  }
}
```

As the plugin uses the web debug toolbar as its main user interface, be sure to have it enabled (which the case in the development environment by default). When enabled, a new menu named “Functional Test” is made available. In this panel, you can start recording a session by clicking on the “Activate” link, and reset the current session by clicking on “Reset”. When you are done, copy and paste the code from the textarea to a test file and start customizing it.

## Run your Test Suite faster

When you have a large suite of tests, it can be very time consuming to launch all tests every time you make a change, especially if some tests fail. Each time you fix a test, you should run the whole test suite again to ensure that you have not broken other tests. But until the failed tests are fixed, there is no point in re-executing all the other tests. To speed up this process, the `test:all` task has an `--only-failed` (`-f` as a shortcut) option that forces the task to only re-execute tests that failed during the previous run:

Listing 3-23
```
$ php symfony test:all --only-failed
```

On first execution, all tests are run as usual. But for subsequent test runs, only tests that failed last time are executed. As you fix your code, some tests will pass, and will be removed from subsequent runs. When all tests pass again, the full test suite is run... you can then rinse and repeat.

Chapter 4

# Emails

*by Fabien Potencier*

Sending emails with symfony is simple and powerful, thanks to the usage of the Swift Mailer[49] 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 4-1*

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 documentation[50].

## Sending Emails from an Action

From an action, retrieving the mailer instance is made simple with the `getMailer()` shortcut method:

```
$mailer = $this->getMailer();
```

*Listing 4-2*

### The Fastest Way

Sending an email is then as simple as using the `sfMailer::composeAndSend()` method:

49. `http://www.swiftmailer.org/`
50. `http://www.swiftmailer.org/docs`

Listing 4-3
```
$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 4-4
```
$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 4-5
```
$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 `sfMailer::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 4-6
```
// 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 4-7
```
$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”[51] and “Message Headers”[52] 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 4-8*

## 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 4-9*

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 4-10*

51. http://swiftmailer.org/docs/messages
52. 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:

Listing 4-11
```
# Schema configuration in schema.yml
MailMessage:
 actAs: { Timestampable: ~ }
 columns:
   message: { type: blob, notnull: true }
```

Listing 4-12
```
# configuration in factories.yml
mailer:
  class: sfMailer
  param:
    delivery_strategy: spool
    spool_class:       Swift_DoctrineSpool
    spool_arguments:   [ MailMessage, message, getSpooledMessages ]
```

And the same configuration for a Propel spool:

Listing 4-13
```
# Schema configuration in schema.yml
mail_message:
  message:    { type: blob, required: true }
  created_at: ~
```

Listing 4-14
```
# configuration in factories.yml
dev:
  mailer:
    param:
      delivery_strategy: spool
      spool_class:       Swift_PropelSpool
      spool_arguments:   [ MailMessage, message, getSpooledMessages ]
```

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):

Listing 4-15
```
$ php symfony project:send-emails
```

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 4-16*
```
$this->getMailer()->sendNextImmediately()->send($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 4-17*
```
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"[53] 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()`:

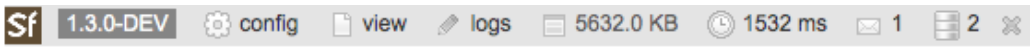
```
$this->getMailer()->sendNextImmediately()->send($message);
```

*Listing 4-18*

## 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.

53. `http://swiftmailer.org/docs/transport-types`


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 4-19*

```
$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 4-20*

```
$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:

Listing 4-21

```
$browser->
  get('/foo')->
  with('mailer')->begin()->
    hasSent(2)->
    withMessage('foo@example.com')->
    checkHeader('Subject', '/Subject/')->
    checkBody('/Body/')->
  end()
;
```

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:

Listing 4-22

```
$browser->
  get('/foo')->
  with('mailer')->
  debug()
;
```

# 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:

Listing 4-23

```
// 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('...')
    ;
  }
}
```

Sending a message from an action, or from anywhere else for that matter, is simple a matter of instantiating the right message class:

Listing 4-24
```
$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 4-25
```
// 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 4-26
```
// 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:

Listing 4-27

```
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
```

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:

Listing 4-28

```
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
  }
}
```

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):

Listing 4-29

```
public function configureMailer(sfEvent $event)
{
  $mailer = $event->getSubject();

  $plugin = new Swift_Plugins_ThrottlerPlugin(
```

```
    100, Swift_Plugins_ThrottlerPlugin::MESSAGES_PER_MINUTE
  );

  $mailer->registerPlugin($plugin);
}
```

Some plugins should be triggered when the emails are actually sent out (AntiFlood, BandwithMonitor, Throttler). They should be registered for the realtime transport only. Otherwise they would be triggered when email is queued rather than when the queue is flushed.

In order for those plugins to have the expected behavior, the code below should always be used to register the plugins when a spool is used. Note than this code is still valid when the mailer doesn't use a spool.

Listing 4-30
```
public function configureMailer(sfEvent $event)
{
  $mailer = $event->getSubject();
  $transport = $mailer->getRealtimeTransport();

  $transport->registerPlugin(new Swift_Plugins_ThrottlerPlugin(
    100, Swift_Plugins_ThrottlerPlugin::MESSAGES_PER_MINUTE
  ));

  $transport->registerPlugin(new Swift_Plugins_AntiFloodPlugin(30));
}
```

The "Plugins"[54] 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 4-31
```
$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 4-32
```
# for Propel
mail_message:
  message:    { type: blob, required: true }
  created_at: ~
  priority:   { type: integer, default: 3 }

# for Doctrine
MailMessage:
```

54. http://swiftmailer.org/docs/plugins

```
  actAs: { Timestampable: ~ }
  columns:
    message:  { type: blob, notnull: true }
    priority: { type: integer }
```

When sending an email, set the priority header (1 means highest):

Listing 4-33

```
$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 4-34

```
// for Propel
class MailMessage extends BaseMailMessage
{
  public function setMessage($message)
  {
    $msg = unserialize($message);
    $this->setPriority($msg->getPriority());

    return parent::setMessage($message);
  }
}

// for Doctrine
class MailMessage extends BaseMailMessage
{
  public function setMessage($message)
  {
    $msg = unserialize($message);
    $this->priority = $msg->getPriority();

    return $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:

Listing 4-35

```
// 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')
    ;
  }

  // ...
}
```

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:

Listing 4-36
```
spool_arguments: [ MailMessage, message, getSpooledMessages ]
```

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 4-37
```
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 4-38
```
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');

  return parent::_set('message', serialize($msg));
}
```

Chapter 5

# Custom Widgets and Validators

*by Thomas Rabaix*

This chapter explains how to build a custom widget and validator for use in the form framework. It will explain the internals of `sfWidgetForm` and `sfValidator`, as well as how to build both a simple and complex widget.

## Widget and Validator Internals

### `sfWidgetForm` Internals

An object of the `sfWidgetForm` class represents the visual implementation of how related data will be edited. A string value, for example, might be edited with a simple text box or an advanced WYSIWYG editor. In order to be fully configurable, the `sfWidgetForm` class has two important properties: `options` and `attributes`.

- `options`: used to configure the widget (e.g. the database query to be used when creating a list to be used in a select box)
- `attributes`: HTML attributes added to the element upon rendering

Additionally, the `sfWidgetForm` class implements two important methods:

- `configure()`: defines which options are *optional* or *mandatory*. While it is not a good practice to override the constructor, the `configure()` method can be safely overridden.
- `render()`: outputs the HTML for the widget. The method has a mandatory first argument, the HTML widget name, and an optional second argument, the value.

---

An `sfWidgetForm` object does not know anything about its name or its value. The component is responsible only for rendering the widget. The name and the value are managed by an `sfFormFieldSchema` object, which is the link between the data and the widgets.

---

### sfValidatorBase Internals

The `sfValidatorBase` class is the base class of each validator. The `sfValidatorBase::clean()` method is the most important method of this class as it checks if the value is valid depending on the provided options.

Internally, the `clean()` method perform several different actions:

- trims the input value for string values (if specified via the `trim` option)
- checks if the value is empty
- calls the validator's `doClean()` method.

The `doClean()` method is the method which implements the main validation logic. It is not good practice to override the `clean()` method. Instead, always perform any custom logic via the `doClean()` method.

A validator can also be used as a standalone component to check input integrity. For instance, the `sfValidatorEmail` validator will check if the email is valid:

Listing 5-1
```
$v = new sfValidatorEmail();

try
{
  $v->clean($request->getParameter("email"));
}
catch(sfValidatorError $e)
{
  $this->forward404();
}
```

When a form is bound to the request values, the `sfForm` object keeps references to the original (dirty) values and the validated (clean) values. The original values are used when the form is redrawn, while the cleaned values are used by the application (e.g. to save the object).

### The `options` Attribute

Both the `sfWidgetForm` and `sfValidatorBase` objects have a variety of options: some are optional while others are mandatory. These options are defined inside each class's `configure()` method via:

- `addOption($name, $value)`: defines an option with a name and a default value
- `addRequiredOption($name)`: defines a mandatory option

These two methods are very convenient as they ensure that dependency values are correctly passed to the validator or the widget.

## Building a Simple Widget and Validator

This section will explain how to build a simple widget. This particular widget will be called a "Trilean" widget. The widget will display a select box with three choices: `No`, `Yes` and `Null`.

Listing 5-2
```
class sfWidgetFormTrilean extends sfWidgetForm
{
  public function configure($options = array(), $attributes = array())
  {

    $this->addOption('choices', array(
      0 => 'No',
      1 => 'Yes',
      'null' => 'Null'
    ));
  }
```

```
  public function render($name, $value = null, $attributes = array(),
$errors = array())
  {
    $value = $value === null ? 'null' : $value;

    $options = array();
    foreach ($this->getOption('choices') as $key => $option)
    {
      $attributes = array('value' => self::escapeOnce($key));
      if ($key == $value)
      {
        $attributes['selected'] = 'selected';
      }

      $options[] = $this->renderContentTag(
        'option',
        self::escapeOnce($option),
        $attributes
      );
    }

    return $this->renderContentTag(
      'select',
      "\n".implode("\n", $options)."\n",
      array_merge(array('name' => $name), $attributes
    ));
  }
}
```

The `configure()` method defines the option values list via the `choices` option. This array can be redefined (i.e. to change the associated label of each value). There is no limit to the number of option a widget can define. The base widget class, however, declares a few standard options, which function like de-facto reserved options:

- `id_format`: the id format, default is '%s'
- `is_hidden`: boolean value to define if the widget is a hidden field (used by `sfForm::renderHiddenFields()` to render all hidden fields at once)
- `needs_multipart`: boolean value to define if the form tag should include the multipart option (i.e. for file uploads)
- `default`: The default value that should be used to render the widget if no value is provided
- `label`: The default widget label

The `render()` method generates the corresponding HTML for a select box. The method calls the built-in `renderContentTag()` function to help render HTML tags.

The widget is now complete. Let's create the corresponding validator:

*Listing 5-3*

```
class sfValidatorTrilean extends sfValidatorBase
{
  protected function configure($options = array(), $messages = array())
  {
    $this->addOption('true_values', array('true', 't', 'yes', 'y', 'on',
'1'));
    $this->addOption('false_values', array('false', 'f', 'no', 'n', 'off',
'0'));
```

```
    $this->addOption('null_values', array('null', null));
  }

  protected function doClean($value)
  {
    if (in_array($value, $this->getOption('true_values')))
    {
      return true;
    }

    if (in_array($value, $this->getOption('false_values')))
    {
      return false;
    }

    if (in_array($value, $this->getOption('null_values')))
    {
      return null;
    }

    throw new sfValidatorError($this, 'invalid', array('value' => $value));
  }

  public function isEmpty($value)
  {
    return false;
  }
}
```

The `sfValidatorTrilean` validator defines three options in the `configure()` method. Each option is a set of valid values. As these are defined as options, the developer can customize the values depending on the specification.

The `doClean()` method checks if the value matches a set a valid values and returns the cleaned value. If no value is matched, the method will raise an `sfValidatorError` which is the standard validation error in the form framework.

The last method, `isEmpty()`, is overridden as the default behavior of this method is to return `true` if `null` is provided. As the current widget allows `null` as a valid value, the method must always return `false`.

---

If `isEmpty()` returns true, the `doClean()` method will never be called.

---

While this widget was fairly straightforward, it introduced some important base features that will be needed as we go further. The next section will create a more complex widget with multiple fields and JavaScript interaction.

# The Google Address Map Widget

In this section, we are going to build a complex widget. New methods will be introduced and the widget will have some JavaScript interaction as well. The widget will be called “GMAW”: “Google Map Address Widget”.

What do we want to achieve? The widget should provide an easy way for the end user to add an address. By using an input text field and with google’s map services we can achieve this goal.


Use case 1:

- The user types an address.
- The user clicks the "lookup" button.
- The latitude and longitude hidden fields are updated and a new marker is created on the map. The marker is positioned at the location of the address. If the Google Geocoding service cannot find the address an error message will popup.

Use case 2:

- The user clicks on the map.
- The latitude and longitude hidden fields are updated.
- Reverse lookup is used to find the address.

*The following fields need to be posted and handled by the form:*

- `latitude`: float, between 90 and -90
- `longitude`: float, between 180 and -180
- `address`: string, plain text only

The widget's functional specifications have just been defined, now let's define the technical tools and their scopes:

- Google map and Geocoding services: displays the map and retrieves address information
- jQuery: adds JavaScript interactions between the form and the field
- sfForm: draws the widget and validates the inputs

## `sfWidgetFormGMapAddress` Widget

As a widget is the visual representation of data, the `configure()` method of the widget must have different options to tweak the Google map or modify the styles of each element. One of the most important options is the `template.html` option, which defines how all elements are ordered. When building a widget it is very important to think about reusability and extensibility.

Another important thing is the external assets definition. An `sfWidgetForm` class can implement two specific methods:

- `getJavascripts()` must return an array of JavaScript files;
- `getStylesheets()` must return an array of stylesheet files (where the key is the path and the value the media name).

The current widget only requires some JavaScript to work so no stylesheet is needed. In this case, however, the widget will not handle the initialization of the Google JavaScript, though the widget will make use of the Google geocoding and map services. Instead, it will be the developer's responsibility to include it on the page. The reason behind this is that Google's services may be used by other elements on the page, and not only by the widget.

Let's jump to the code:

*Listing 5-4*

```
class sfWidgetFormGMapAddress extends sfWidgetForm
{
  public function configure($options = array(), $attributes = array())
  {
    $this->addOption('address.options', array('style' => 'width:400px'));

    $this->setOption('default', array(
      'address' => '',
      'longitude' => '2.294359',
      'latitude' => '48.858205'
    ));

    $this->addOption('div.class', 'sf-gmap-widget');
    $this->addOption('map.height', '300px');
    $this->addOption('map.width', '500px');
    $this->addOption('map.style', "");
    $this->addOption('lookup.name', "Lookup");

    $this->addOption('template.html', '
      <div id="{div.id}" class="{div.class}">
        {input.search} <input type="submit" value="{input.lookup.name}"
id="{input.lookup.id}" /> <br />
        {input.longitude}
        {input.latitude}
        <div id="{map.id}"
style="width:{map.width};height:{map.height};{map.style}"></div>
      </div>
    ');

     $this->addOption('template.javascript', '
      <script type="text/javascript">
        jQuery(window).bind("load", function() {
          new sfGmapWidgetWidget({
            longitude: "{input.longitude.id}",
            latitude: "{input.latitude.id}",
            address: "{input.address.id}",
            lookup: "{input.lookup.id}",
            map: "{map.id}"
          });
        })
      </script>
    ');
  }

  public function getJavascripts()
  {
```

```
    return array(
      '/sfFormExtraPlugin/js/sf_widget_gmap_address.js'
    );
  }

  public function render($name, $value = null, $attributes = array(),
$errors = array())
  {
    // define main template variables
    $template_vars = array(
      '{div.id}'             => $this->generateId($name),
      '{div.class}'          => $this->getOption('div.class'),
      '{map.id}'             => $this->generateId($name.'[map]'),
      '{map.style}'          => $this->getOption('map.style'),
      '{map.height}'         => $this->getOption('map.height'),
      '{map.width}'          => $this->getOption('map.width'),
      '{input.lookup.id}'    => $this->generateId($name.'[lookup]'),
      '{input.lookup.name}'  => $this->getOption('lookup.name'),
      '{input.address.id}'   => $this->generateId($name.'[address]'),
      '{input.latitude.id}'  => $this->generateId($name.'[latitude]'),
      '{input.longitude.id}' => $this->generateId($name.'[longitude]'),
    );

    // avoid any notice errors to invalid $value format
    $value = !is_array($value) ? array() : $value;
    $value['address']   = isset($value['address'])   ? $value['address'] :
'';
    $value['longitude'] = isset($value['longitude']) ? $value['longitude']
: '';
    $value['latitude']  = isset($value['latitude'])  ? $value['latitude']
: '';

    // define the address widget
    $address = new sfWidgetFormInputText(array(),
$this->getOption('address.options'));
    $template_vars['{input.search}'] = $address->render($name.'[address]',
$value['address']);

    // define the longitude and latitude fields
    $hidden = new sfWidgetFormInputHidden;
    $template_vars['{input.longitude}'] =
$hidden->render($name.'[longitude]', $value['longitude']);
    $template_vars['{input.latitude}']  =
$hidden->render($name.'[latitude]', $value['latitude']);

    // merge templates and variables
    return strtr(

$this->getOption('template.html').$this->getOption('template.javascript'),
      $template_vars
    );
  }
}
```

The widget uses the `generateId()` method to generate the `id` of each element. The `$name` variable is defined by the `sfFormFieldSchema`, so the `$name` variable is composed of the name form, any nested widget schema names and the name of the widget as defined in the form `configure()`.

For instance, if the form name is `user`, the nested schema name is `location` and the widget name is `address`, the final `name` will be `user[location][address]` and the `id` will be `user_location_address`. In other words, `$this->generateId($name.'[latitude]')` will generate a valid and unique `id` for the `latitude` field.

The different element `id` attributes are quite important as there are passed to the JavaScript block (via the `template.js` variable), so the JavaScript can properly handle the different elements.

The `render()` method also instantiates two inner widgets: an `sfWidgetFormInputText` widget, which is used to render the address field, and an `sfWidgetFormInputHidden` widget, which is used to render the hidden fields.

The widget can be quickly tested with this small piece of code:

*Listing 5-5*

```
$widget = new sfWidgetFormGMapAddress();
echo $widget->render('user[location][address]', array(
  'address' => '151 Rue montmartre, 75002 Paris',
  'longitude' => '2.294359',
  'latitude' => '48.858205'
));
```

The output result is:

*Listing 5-6*

```
<div id="user_location_address" class="sf-gmap-widget">
  <input style="width:400px" type="text"
name="user[location][address][address]" value="151 Rue montmartre, 75002
Paris" id="user_location_address_address" />
  <input type="submit" value="Lookup"  id="user_location_address_lookup"
/> <br />
  <input type="hidden" name="user[location][address][longitude]"
value="2.294359" id="user_location_address_longitude" />
  <input type="hidden" name="user[location][address][latitude]"
value="48.858205" id="user_location_address_latitude" />
  <div id="user_location_address_map"
style="width:500px;height:300px;"></div>
</div>

<script type="text/javascript">
  jQuery(window).bind("load", function() {
    new sfGmapWidgetWidget({
      longitude: "user_location_address_longitude",
      latitude: "user_location_address_latitude",
      address: "user_location_address_address",
      lookup: "user_location_address_lookup",
      map: "user_location_address_map"
    });
  })
</script>
```

The JavaScript part of the widget takes the different `id` attributes and binds jQuery listeners to them so that certain JavaScript is triggered when actions are performed. The JavaScript updates the hidden fields with the longitude and latitude provided by the google geocoding service.

The JavaScript object has a few interesting methods:

- `init()`: the method where all variables are initialized and events bound to different inputs
- `lookupCallback()`: a *static* method used by the geocoder method to lookup the address provided by the user
- `reverseLookupCallback()`: is another *static* method used by the geocoder to convert the given longitude and latitude into a valid address.

The final JavaScript code can be viewed in Appendix A.

Please refer to the Google map documentation for more details on the functionality of the Google maps API[55].

### `sfValidatorGMapAddress` Validator

The `sfValidatorGMapAddress` class extends `sfValidatorBase` which already performs one validation: specifically, if the field is set as required then the value cannot be `null`. Thus, `sfValidatorGMapAddress` need only validate the different values: `latitude`, `longitude` and `address`. The `$value` variable should be an array, but as the user input should not be trusted, the validator checks for the presence of all keys so that the inner validators are passed valid values.

*Listing 5-7*

```
class sfValidatorGMapAddress extends sfValidatorBase
{
  protected function doClean($value)
  {
    if (!is_array($value))
    {
      throw new sfValidatorError($this, 'invalid');
    }

    try
    {
      $latitude = new sfValidatorNumber(array( 'min' => -90, 'max' => 90,
'required' => true ));
      $value['latitude'] = $latitude->clean(isset($value['latitude']) ?
$value['latitude'] : null);

      $longitude = new sfValidatorNumber(array( 'min' => -180, 'max' =>
180, 'required' => true ));
      $value['longitude'] = $longitude->clean(isset($value['longitude']) ?
$value['longitude'] : null);

      $address = new sfValidatorString(array( 'min_length' => 10,
'max_length' => 255, 'required' => true ));
      $value['address'] = $address->clean(isset($value['address']) ?
$value['address'] : null);
    }
    catch(sfValidatorError $e)
    {
      throw new sfValidatorError($this, 'invalid');
    }

    return $value;
  }
}
```

55. http://code.google.com/apis/maps/

A validator always raises an `sfValidatorError` exception when a value is not valid. That's why the validation is surrounded by a `try/catch` block. In this validator, the validator re-throws a new `invalid` exception, which equates to an `invalid` validation error on the `sfValidatorGMapAddress` validator.

## Testing

Why is testing important? The validator is the glue between the user input and the application. If the validator is flawed, the application is vulnerable. Fortunately, symfony comes with `lime` which is a testing library that is very easy to use.

How can we test the validator? As stated before, a validator raises an exception on a validation error. The test can send valid and invalid values to the validator and check to see that the exception is thrown in the correct circumstances.

*Listing 5-8*

```
$t = new lime_test(7, new lime_output_color());

$tests = array(
  array(false, '', 'empty value'),
  array(false, 'string value', 'string value'),
  array(false, array(), 'empty array'),
  array(false, array('address' => 'my awesome address'), 'incomplete
address'),
  array(false, array('address' => 'my awesome address', 'latitude' =>
'String', 'longitude' => 23), 'invalid values'),
  array(false, array('address' => 'my awesome address', 'latitude' => 200,
'longitude' => 23), 'invalid values'),
  array(true, array('address' => 'my awesome address', 'latitude' =>
'2.294359', 'longitude' => '48.858205'), 'valid value')
);

$v = new sfValidatorGMapAddress;

$t->diag("Testing sfValidatorGMapAddress");

foreach($tests as $test)
{
  list($validity, $value, $message) = $test;

  try
  {
    $v->clean($value);
    $catched = false;
  }
  catch(sfValidatorError $e)
  {
    $catched = true;
  }

  $t->ok($validity != $catched, '::clean() '.$message);
}
```

When the `sfForm::bind()` method is called, the form executes the `clean()` method of each validator. This test reproduces this behavior by instantiating the `sfValidatorGMapAddress` validator directly and testing different values.

## Final Thoughts

The most common mistake when creating a widget is to be overly focused on how the information will be stored in the database. The form framework is simply a data container and validation framework. Therefore, a widget must only manage its related information. If the data is valid then the different cleaned values can then be used by the model or in the controller.

Chapter 6

# Advanced Forms

*by Ryan Weaver, Fabien Potencier*

Symfony's form framework equips the developer with the tools necessary to easily render and validate form data in an object-oriented matter. Thanks to the `sfFormDoctrine` and `sfFormPropel` classes offered by each ORM, the form framework can easily render and save forms that relate closely to the data layer.

Real-world situations, however, often require the developer to customize and extend forms. In this chapter we'll present and solve several common, but challenging form problems. We'll also dissect the `sfForm` object and remove some of its mystery.

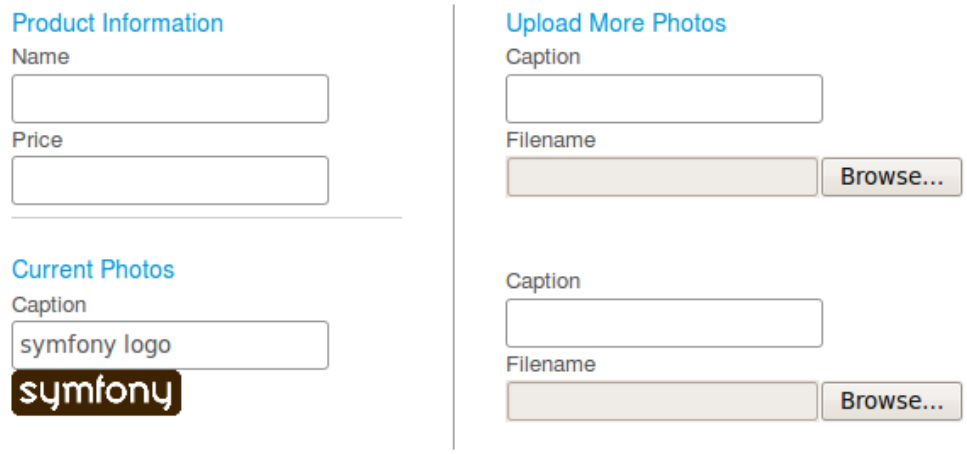
## Mini-Project: Products & Photos

The first problem revolves around editing an individual product and an unlimited number of photos for that product. The user must be able to edit both the Product and the Product's Photos on the same form. We'll also need to allow the user to upload up to two new Product Photos at a time. Here is a possible schema:

*Listing 6-1*

```
Product:
  columns:
    name:           { type: string(255), notnull: true }
    price:          { type: decimal, notnull: true }

ProductPhoto:
  columns:
    product_id:     { type: integer }
    filename:       { type: string(255) }
    caption:        { type: string(255), notnull: true }
  relations:
    Product:
      alias:        Product
      foreignType:  many
      foreignAlias: Photos
      onDelete:     cascade
```

When completed, our form will look something like this:




# Learn more by doing the Examples

The best way to learn advanced techniques is to follow along and test the examples step by step. Thanks to the `--installer` feature of symfony *(page 31)*, we provide a simple way for you to create a working project with a ready to be used SQLite database, the Doctrine database schema, some fixtures, a `frontend` application, and a `product` module to work with. Download the installer script[56] and run the following command to create the symfony project:

```
$ php symfony generate:project advanced_form --installer=/path/to/
advanced_form_installer.php
```

*Listing 6-2*

This command creates a fully-working project with the database schema we have introduced in the previous section.

In this chapter, the file paths are for a symfony project running with Doctrine as generated by the previous task.

# Basic Form Setup

Because the requirements involve changes to two different models (`Product` and `ProductPhoto`), the solution will need to incorporate two different symfony forms (`ProductForm` and `ProductPhotoForm`). Fortunately, the form framework can easily combine multiple forms into one via `sfForm::embedForm()`. First, setup the `ProductPhotoForm` independently. In this example, let's use the `filename` field as a file upload field:

```
// lib/form/doctrine/ProductPhotoForm.class.php
public function configure()
{
  $this->useFields(array('filename', 'caption'));

  $this->setWidget('filename', new sfWidgetFormInputFile());
  $this->setValidator('filename', new sfValidatorFile(array(
```

*Listing 6-3*

56. `http://www.symfony-project.org/images/more-with-symfony/advanced_form_installer.php.src`

```
    'mime_types' => 'web_images',
    'path' => sfConfig::get('sf_upload_dir').'/products',
  )));
}
```

For this form, both the `caption` and `filename` fields are automatically required, but for different reasons. The `caption` field is required because the related column in the database schema has been defined with a `notnull` property set to `true`. The `filename` field is required by default because a validator object defaults to `true` for the `required` option.

`sfForm::useFields()` is a new function to symfony 1.3 that allows the developer to specify exactly which fields the form should use and in which order they should be displayed. All other non-hidden fields are removed from the form.

So far we've done nothing more than ordinary form setup. Next, we'll combine the forms into one.

# Embedding Forms

By using `sfForm::embedForm()`, the independent `ProductForm` and `ProductPhotoForms` can be combined with very little effort. The work is always done in the *main* form, which in this case is `ProductForm`. The requirements call for the ability to upload up to two product photos at once. To accomplish this, embed two `ProductPhotoForm` objects into `ProductForm`:

Listing 6-4
```
// lib/form/doctrine/ProductForm.class.php
public function configure()
{
  $subForm = new sfForm();
  for ($i = 0; $i < 2; $i++)
  {
    $productPhoto = new ProductPhoto();
    $productPhoto->Product = $this->getObject();

    $form = new ProductPhotoForm($productPhoto);

    $subForm->embedForm($i, $form);
  }
  $this->embedForm('newPhotos', $subForm);
}
```

If you point your browser to the `product` module, you now have the ability to upload two `ProductPhotos` as well as modify the `Product` object itself. Symfony automatically saves the new `ProductPhoto` objects and links them to the corresponding `Product` object. Even the file upload, defined in `ProductPhotoForm`, executes normally.

Check that the records are saved correctly in the database:

Listing 6-5
```
$ php symfony doctrine:dql --table "FROM Product"
$ php symfony doctrine:dql --table "FROM ProductPhoto"
```

In the `ProductPhoto` table, you will notice the filenames of the photos. Everything is working as expected if you can find files with the same names as the ones in the database in the `web/uploads/products/` directory.

Because the filename and caption fields are required in `ProductPhotoForm`, validation of the main form will always fail unless the user is uploading two new photos. Keep reading to learn how to fix this problem.

# Refactoring

Even if the previous form works as expected, it would be better to refactor the code a bit to ease testing and to allow the code to be easily reused.

First, let's create a new form that represents a collection of `ProductPhotoForm`s, based on the code we have already written:

Listing 6-6

```
// lib/form/doctrine/ProductPhotoCollectionForm.class.php
class ProductPhotoCollectionForm extends sfForm
{
  public function configure()
  {
    if (!$product = $this->getOption('product'))
    {
      throw new InvalidArgumentException('You must provide a product
object.');
    }

    for ($i = 0; $i < $this->getOption('size', 2); $i++)
    {
      $productPhoto = new ProductPhoto();
      $productPhoto->Product = $product;

      $form = new ProductPhotoForm($productPhoto);

      $this->embedForm($i, $form);
    }
  }
}
```

This form needs two options:

- `product`: The product for which to create a collection of `ProductPhotoForm`s;
- `size`: The number of `ProductPhotoForm`s to create (default to two).

You can now change the configure method of `ProductForm` to read as follows:

Listing 6-7

```
// lib/form/doctrine/ProductForm.class.php
public function configure()
{
  $form = new ProductPhotoCollectionForm(null, array(
    'product' => $this->getObject(),
    'size'    => 2,
  ));

  $this->embedForm('newPhotos', $form);
}
```

# Dissecting the sfForm Object

In the most basic sense, a web form is a collection of fields that are rendered and submitted back to the server. In the same light, the `sfForm` object is essentially an array of form *fields*. While `sfForm` manages the process, the individual fields are responsible for defining how each will be rendered and validated.

In symfony, each form *field* is defined by two different objects:

- A *widget* that outputs the form field's XHTML markup;
- A *validator* that cleans and validates the submitted field data.

In symfony, a *widget* is defined as any object whose sole job is to output XHTML markup. While most commonly used with forms, a widget object could be created to output any markup.

## A Form is an Array

Recall that the `sfForm` object is "essentially an array of form *fields*." To be more precise, `sfForm` houses both an array of widgets and an array of validators for all of the fields of the form. These two arrays, called `widgetSchema` and `validatorSchema` are properties of the `sfForm` class. In order to add a field to a form, we simply add the field's widget to the `widgetSchema` array and the field's validator to the `validatorSchema` array. For example, the following code would add an `email` field to a form:

*Listing 6-8*

```
public function configure()
{
  $this->widgetSchema['email'] = new sfWidgetFormInputText();
  $this->validatorSchema['email'] = new sfValidatorEmail();
}
```

The `widgetSchema` and `validatorSchema` arrays are actually special classes called `sfWidgetFormSchema` and `sfValidatorSchema` that implement the `ArrayAccess` interface.

## Dissecting the `ProductForm`

As the `ProductForm` class ultimately extends `sfForm`, it too houses all of its widgets and validators in `widgetSchema` and `validatorSchema` arrays. Let's look at how each array is organized in the finished `ProductForm` object.

*Listing 6-9*

```
widgetSchema    => array
(
  [id]          => sfWidgetFormInputHidden,
  [name]        => sfWidgetFormInputText,
  [price]       => sfWidgetFormInputText,
  [newPhotos]   => array(
    [0]           => array(
      [id]          => sfWidgetFormInputHidden,
      [filename]    => sfWidgetFormInputFile,
      [caption]     => sfWidgetFormInputText,
    ),
    [1]           => array(
      [id]          => sfWidgetFormInputHidden,
```

```
      [filename]    => sfWidgetFormInputFile,
      [caption]     => sfWidgetFormInputText,
    ),
  ),
)

validatorSchema => array
(
  [id]          => sfValidatorDoctrineChoice,
  [name]        => sfValidatorString,
  [price]       => sfValidatorNumber,
  [newPhotos]   => array(
    [0]           => array(
      [id]          => sfValidatorDoctrineChoice,
      [filename]    => sfValidatorFile,
      [caption]     => sfValidatorString,
    ),
    [1]           => array(
      [id]          => sfValidatorDoctrineChoice,
      [filename]    => sfValidatorFile,
      [caption]     => sfValidatorString,
    ),
  ),
)
```

Just as `widgetSchema` and `validatorSchema` are actually objects that behave as arrays, the above arrays defined by the keys `newPhotos`, `0`, and `1` are also `sfWidgetSchema` and `sfValidatorSchema` objects.

As expected, basic fields (`id`, `name` and `price`) are represented at the first level of each array. In a form that embeds no other forms, both the `widgetSchema` and `validatorSchema` arrays have just one level, representing the basic fields on the form. The widgets and validators of any embedded forms are represented as child arrays in `widgetSchema` and `validatorSchema` as seen above. The method that manages this process is explained next.

## Behind `sfForm::embedForm()`

Keep in mind that a form is composed of an array of widgets and an array of validators. Embedding one form into another essentially means that the widget and validator arrays of one form are added to the widget and validator arrays of the main form. This is entirely accomplished via `sfForm::embedForm()`. The result is always a multi-dimensional addition to the `widgetSchema` and `validatorSchema` arrays as seen above.

Below, we'll discuss the setup of `ProductPhotoCollectionForm`, which binds individual `ProductPhotoForm` objects into itself. This middle form acts as a "wrapper" form and helps with overall form organization. Let's begin with the following code from `ProductPhotoCollectionForm::configure()`:

*Listing 6-10*

```
$form = new ProductPhotoForm($productPhoto);
$this->embedForm($i, $form);
```

The `ProductPhotoCollectionForm` form itself begins as a new `sfForm` object. As such, its `widgetSchema` and `validatorSchema` arrays are empty.

*Listing 6-11*

```
widgetSchema    => array()
validatorSchema => array()
```

Each `ProductPhotoForm`, however, is already prepared with three fields (`id`, `filename`, and `caption`) and three corresponding items in its `widgetSchema` and `validatorSchema` arrays.

*Listing 6-12*

```
widgetSchema    => array
(
  [id]            => sfWidgetFormInputHidden,
  [filename]      => sfWidgetFormInputFile,
  [caption]       => sfWidgetFormInputText,
)

validatorSchema => array
(
  [id]            => sfValidatorDoctrineChoice,
  [filename]      => sfValidatorFile,
  [caption]       => sfValidatorString,
)
```

The `sfForm::embedForm()` method simply adds the `widgetSchema` and `validatorSchema` arrays from each `ProductPhotoForm` to the `widgetSchema` and `validatorSchema` arrays of the empty `ProductPhotoCollectionForm` object.

When finished, the `widgetSchema` and `validatorSchema arrays` of the wrapper form (`ProductPhotoCollectionForm`) are multi-level arrays that hold the widgets and validators from both `ProductPhotoForm`s.

*Listing 6-13*

```
widgetSchema    => array
(
  [0]             => array
  (
    [id]            => sfWidgetFormInputHidden,
    [filename]      => sfWidgetFormInputFile,
    [caption]       => sfWidgetFormInputText,
  ),
  [1]             => array
  (
    [id]            => sfWidgetFormInputHidden,
    [filename]      => sfWidgetFormInputFile,
    [caption]       => sfWidgetFormInputText,
  ),
)

validatorSchema => array
(
  [0]             => array
  (
    [id]            => sfValidatorDoctrineChoice,
    [filename]      => sfValidatorFile,
    [caption]       => sfValidatorString,
  ),
  [1]             => array
  (
    [id]            => sfValidatorDoctrineChoice,
    [filename]      => sfValidatorFile,
    [caption]       => sfValidatorString,
```

```
  ),
)
```

In the final step of our process, the resulting wrapper form, `ProductPhotoCollectionForm`, is embedded directly into `ProductForm`. This occurs inside `ProductForm::configure()`, which takes advantage of all the work that was done inside `ProductPhotoCollectionForm`:

Listing 6-14

```
$form = new ProductPhotoCollectionForm(null, array(
  'product' => $this->getObject(),
  'size'    => 2,
));

$this->embedForm('newPhotos', $form);
```

This gives us the final `widgetSchema` and `validatorSchema` array structure seen above. Notice that the `embedForm()` method is very similar to the simple act of combining the `widgetSchema` and `validatorSchema` arrays manually:

Listing 6-15

```
$this->widgetSchema['newPhotos'] = $form->getWidgetSchema();
$this->validatorSchema['newPhotos'] = $form->getValidatorSchema();
```

# Rendering Embedded Forms in the View

The current `_form.php` template of the `product` module looks like the following:

Listing 6-16

```
// apps/frontend/module/product/templates/_form.php
<!-- ... -->

<tbody>
  <?php echo $form ?>
</tbody>

<!-- ... -->
```

The `<?php echo $form ?>` statement is the simplest way to display a form, even the most complex ones. It is of great help when prototyping, but as soon as you want to change the layout, you need to replace it with your own display logic. Remove this line now as we will replace it in this section.

The most important thing to understand when rendering embedded forms in the view is the organization of the multi-level `widgetSchema` array explained in the previous sections. For this example, let's begin by rendering the basic `name` and `price` fields from the `ProductForm` in the view:

Listing 6-17

```
// apps/frontend/module/product/templates/_form.php
<?php echo $form['name']->renderRow() ?>

<?php echo $form['price']->renderRow() ?>

<?php echo $form->renderHiddenFields() ?>
```

As its name implies, the `renderHiddenFields()` renders all the hidden fields of the form.

The actions code was purposefully not shown here because it requires no special attention. Have a look at the `apps/frontend/modules/product/actions/actions.class.php` actions file. It looks like any normal CRUD and can be generated automatically via the `doctrine:generate-module` task.

As we've already learned, the `sfForm` class houses the `widgetSchema` and `validatorSchema` arrays that define our fields. Moreover, the `sfForm` class implements the native PHP 5 `ArrayAccess` interface, meaning we can directly access fields of the form by using the array key syntax seen above.

To output the fields, you can simply access them directly and call the `renderRow()` method. But what type of object is `$form['name']`? While you might expect the answer to be the `sfWidgetFormInputText` widget for the `name` field, the answer is actually something slightly different.

## Rendering each Form Field with `sfFormField`

By using the `widgetSchema` and `validatorSchema` arrays defined in each form class, `sfForm` automatically generates a third array called `sfFormFieldSchema`. This array contains a special object for each field that acts as a helper class responsible for the field's output. The object, of type `sfFormField`, is a combination of each field's widget and validator and is automatically created.

*Listing 6-18*
```
<?php echo $form['name']->renderRow() ?>
```

In the above snippet, `$form['name']` is an `sfFormField` object, which houses the `renderRow()` method along with several other useful rendering functions.

## sfFormField Rendering Methods

Each `sfFormField` object can be used to easily render every aspect of the field that it represents (e.g. the field itself, the label, error messages, etc.). Some of the useful methods inside `sfFormField` include the following. Other can be found via the symfony 1.3 API[57].

- `sfFormField->render()`: Renders the form field (e.g. `input`, `select`) with the correct value using the field's widget object.
- `sfFormField->renderError()`: Renders any validation errors on the field using the field's validator object.
- `sfFormField->renderRow()`: All-encompassing: renders the label, the form field, the error and the help message inside an XHTML markup wrapper.

In reality, each rendering function of the `sfFormField` class also uses information from the form's `widgetSchema` property (the `sfWidgetFormSchema` object that houses all of the widgets for the form). This class assists in the generation of each field's `name` and `id` attributes, keeps track of the label for each field, and defines the XHTML markup used with `renderRow()`.

One important thing to note is that the `formFieldSchema` array always mirrors the structure of the form's `widgetSchema` and `validatorSchema` arrays. For example, the `formFieldSchema` array of the completed `ProductForm` would have the following structure, which is the key to rendering each field in the view:

57. `http://www.symfony-project.org/api/1_3/sfFormField`

Listing 6-19

```
formFieldSchema    => array
(
  [id]          => sfFormField
  [name]        => sfFormField,
  [price]       => sfFormField,
  [newPhotos]   => array(
    [0]           => array(
      [id]          => sfFormField,
      [filename]    => sfFormField,
      [caption]     => sfFormField,
    ),
    [1]           => array(
      [id]          => sfFormField,
      [filename]    => sfFormField,
      [caption]     => sfFormField,
    ),
  ),
)
```

### Rendering the New ProductForm

Using the above array as our map, we can easily output the embedded `ProductPhotoForm` fields in the view by locating and rendering the proper `sfFormField` objects:

Listing 6-20

```
// apps/frontend/module/product/templates/_form.php
<?php foreach ($form['newPhotos'] as $photo): ?>
  <?php echo $photo['caption']->renderRow() ?>
  <?php echo $photo['filename']->renderRow() ?>
<?php endforeach; ?>
```

The above block loops twice: once for the `0` form field array and once for the `1` form field array. As seen in the above diagram, the underlying objects of each array are `sfFormField` objects, which we can output like any other fields.

## Saving Object Forms

Under most circumstances, a form will relate directly to one or more database tables and trigger changes to the data in those tables based on the submitted values. Symfony automatically generates a form object for each schema model, which extends either `sfFormDoctrine` or `sfFormPropel` depending on your ORM. Each form class is similar and ultimately allows for submitted values to be easily persisted in the database.

`sfFormObject` is a new class added in symfony 1.3 to handle all of the common tasks of `sfFormDoctrine` and `sfFormPropel`. Each class extends `sfFormObject`, which now manages part of the form-saving process described below.

### The Form Saving Process

In our example, symfony automatically saves both the `Product` information and new `ProductPhoto` objects without any additional effort by the developer. The method that triggers the magic, `sfFormObject::save()`, executes a variety of methods behind the scenes. Understanding this process is key to extending the process in more advanced situations.

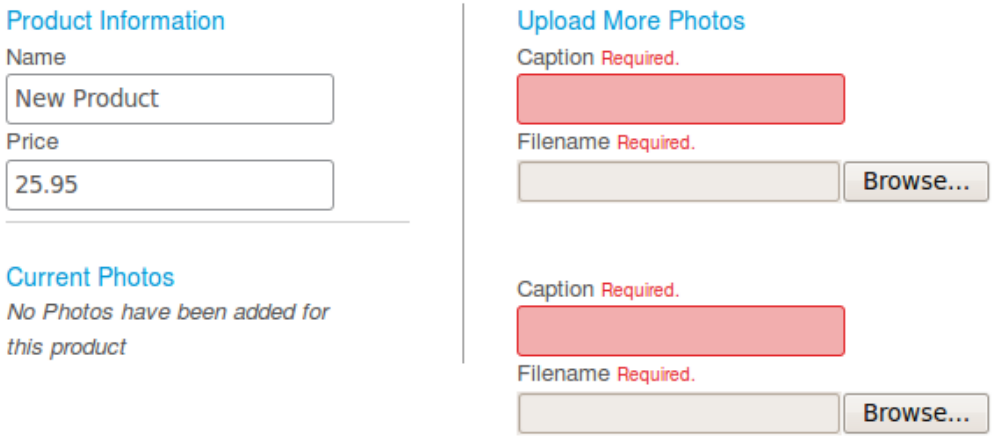
The form saving process consists of a series of internally executed methods, all of which happen after calling `sfFormObject::save()`. The majority of the work is wrapped in the `sfFormObject::updateObject()` method, which is called recursively on all of your embedded forms.

- *sfFormObject::save()*
  - *sfFormObject::doSave()*
    - *sfFormObject::updateObject()*
      1. *sfFormDoctrine::processValues($values)*
         - The *$values* array passed to this method is the raw, bound values array.
         - This method basically allows for any processing to be done on top-level bound values before they are used to update the object.
         - Method *updateXXXColumn()* is called on each field if it exists.
         - Calls *sfFormDoctrine::processUploadedFile()* which transforms all values from upload fields into sfValidatedFile objects.
      2. *sfFormDoctrine::doUpdateObject($values)*
         - The *$values* array is the array returned by *processValues()*.
         - This simply updates the object with the *$values* array.
      3. *sfFormObject::updateObjectEmbeddedForms($values)*
         - Calls *sfFormObject::updateObject()* on each embedded form. In essence, the process (steps 1-3) repeats recursively onto all embedded forms.

The majority of the saving process takes place from within the `sfFormObject::doSave()` method, which is called by `sfFormObject::save()` and wrapped in a database transaction. If you need to modify the saving process itself, `sfFormObject::doSave()` is usually the best place to do it.

# Ignoring Embedded Forms

The current `ProductForm` implementation has one major shortfall. Because the `filename` and `caption` fields are required in `ProductPhotoForm`, validation of the main form will always fail unless the user is uploading two new photos. In other words, the user can't simply change the price of the `Product` without also being required to upload two new photos.




Let's redefine the requirements to include the following. If the user leaves all the fields of a `ProductPhotoForm` blank, that form should be ignored completely. However, if at least one field has data (i.e. `caption` or `filename`), the form should validate and save normally. To

accomplish this, we'll employ an advanced technique involving the use of a custom post validator.

The first step, however, is to modify the `ProductPhotoForm` form to make the `caption` and `filename` fields optional:

Listing 6-21

```
// lib/form/doctrine/ProductPhotoForm.class.php
public function configure()
{
  $this->setValidator('filename', new sfValidatorFile(array(
    'mime_types' => 'web_images',
    'path' => sfConfig::get('sf_upload_dir').'/products',
    'required' => false,
  )));

  $this->validatorSchema['caption']->setOption('required', false);
}
```

In the above code, we have set the `required` option to `false` when overriding the default validator for the `filename` field. Additionally, we have explicitly set the `required` option of the `caption` field to `false`.

Now, let's add the post validator to the `ProductPhotoCollectionForm`:

Listing 6-22

```
// lib/form/doctrine/ProductPhotoCollectionForm.class.php
public function configure()
{
  // ...

  $this->mergePostValidator(new ProductPhotoValidatorSchema());
}
```

A post validator is a special type of validator that validates across all of the submitted values (as opposed to validating the value of a single field). One of the most common post validators is `sfValidatorSchemaCompare` which verifies, for example, that one field is less than another field.

## Creating a Custom Validator

Fortunately, creating a custom validator is actually quite easy. Create a new file, `ProductPhotoValidatorSchema.class.php` and place it in the `lib/validator/` directory (you'll need to create this directory):

Listing 6-23

```
// lib/validator/ProductPhotoValidatorSchema.class.php
class ProductPhotoValidatorSchema extends sfValidatorSchema
{
  protected function configure($options = array(), $messages = array())
  {
    $this->addMessage('caption', 'The caption is required.');
    $this->addMessage('filename', 'The filename is required.');
  }

  protected function doClean($values)
  {
    $errorSchema = new sfValidatorErrorSchema($this);

    foreach($values as $key => $value)
    {
```

```
      $errorSchemaLocal = new sfValidatorErrorSchema($this);

      // filename is filled but no caption
      if ($value['filename'] && !$value['caption'])
      {
        $errorSchemaLocal->addError(new sfValidatorError($this,
'required'), 'caption');
      }

      // caption is filled but no filename
      if ($value['caption'] && !$value['filename'])
      {
        $errorSchemaLocal->addError(new sfValidatorError($this,
'required'), 'filename');
      }

      // no caption and no filename, remove the empty values
      if (!$value['filename'] && !$value['caption'])
      {
        unset($values[$key]);
      }

      // some error for this embedded-form
      if (count($errorSchemaLocal))
      {
        $errorSchema->addError($errorSchemaLocal, (string) $key);
      }
    }

    // throws the error for the main form
    if (count($errorSchema))
    {
      throw new sfValidatorErrorSchema($this, $errorSchema);
    }

    return $values;
  }
}
```

All validators extend `sfValidatorBase` and require only the `doClean()` method. The `configure()` method can also be used to add options or messages to the validator. In this case, two messages are added to the validator. Similarly, additional options can be added via the `addOption()` method.

The `doClean()` method is responsible for cleaning and validating the bound values. The logic of the validator itself is quite simple:

- If a photo is submitted with only the filename or a caption, we throw an error (`sfValidatorErrorSchema`) with the appropriate message;
- If a photo is submitted with no filename and no caption, we remove the values altogether to avoid saving an empty photo;
- If no validation errors have occurred, the method returns the array of cleaned values.

Because the custom validator in this situation is meant to be used as a post validator, the `doClean()` method expects an array of the bound values and returns an array of cleaned values. Custom validators, however, can just as easily be created for individual fields. In that case, the `doClean()` method will expect just one value (the value of the submitted field) and will return just one value.

The last step is to override the `saveEmbeddedForms()` method of `ProductForm` to remove empty photo forms to avoid saving an empty photo in the database (it would otherwise throws an exception as the `caption` column is required):

*Listing 6-24*

```
public function saveEmbeddedForms($con = null, $forms = null)
{
  if (null === $forms)
  {
    $photos = $this->getValue('newPhotos');
    $forms = $this->embeddedForms;
    foreach ($this->embeddedForms['newPhotos'] as $name => $form)
    {
      if (!isset($photos[$name]))
      {
        unset($forms['newPhotos'][$name]);
      }
    }
  }

  return parent::saveEmbeddedForms($con, $forms);
}
```

# Easily Embedding Doctrine-Related Forms

New to symfony 1.3 is the `sfFormDoctrine::embedRelation()` function which allows the developer to embed n-to-many relationship into a form automatically. Suppose, for example, that in addition to allowing the user to upload two new `ProductPhotos`, we also want to allow the user to modify the existing `ProductPhoto` objects related to this `Product`.

Next, use the `embedRelation()` method to add one additional `ProductPhotoForm` object for each existing `ProductPhoto` object:

*Listing 6-25*

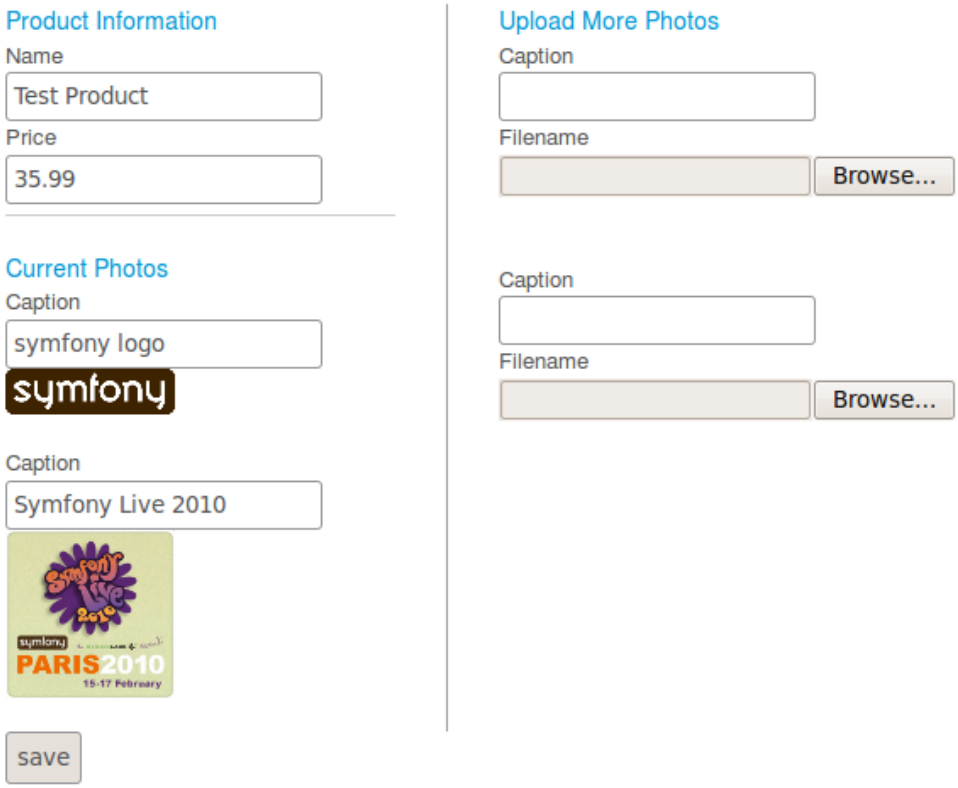
```
// lib/form/doctrine/ProductForm.class.php
public function configure()
{
  // ...

  $this->embedRelation('Photos');
}
```

Internally, `sfFormDoctrine::embedRelation()` does almost exactly what we did manually to embed our two new `ProductPhotoForm` objects. If two `ProductPhoto` relations exist already, then the resulting `widgetSchema` and `validatorSchema` of our form would take the following shape:

*Listing 6-26*

```
widgetSchema    => array
(
  [id]          => sfWidgetFormInputHidden,
  [name]        => sfWidgetFormInputText,
```

```
  [price]       => sfWidgetFormInputText,
  [newPhotos]   => array(...)
  [Photos]      => array(
    [0]           => array(
      [id]          => sfWidgetFormInputHidden,
      [caption]     => sfWidgetFormInputText,
    ),
    [1]           => array(
      [id]          => sfWidgetFormInputHidden,
      [caption]     => sfWidgetFormInputText,
    ),
  ),
)

validatorSchema => array
(
  [id]          => sfValidatorDoctrineChoice,
  [name]        => sfValidatorString,
  [price]       => sfValidatorNumber,
  [newPhotos]   => array(...)
  [Photos]      => array(
    [0]           => array(
      [id]          => sfValidatorDoctrineChoice,
      [caption]     => sfValidatorString,
    ),
    [1]           => array(
      [id]          => sfValidatorDoctrineChoice,
      [caption]     => sfValidatorString,
    ),
  ),
)
```



The next step is to add code to the view that will render the new embedded *Photo* forms:

Listing 6-27

```
// apps/frontend/module/product/templates/_form.php
<?php foreach ($form['Photos'] as $photo): ?>
  <?php echo $photo['caption']->renderRow() ?>
  <?php echo $photo['filename']->renderRow(array('width' => 100)) ?>
<?php endforeach; ?>
```

This snippet is exactly the one we used earlier to embed the new photo forms.

The last step is to convert the file upload field by one which allows the user to see the current photo and to change it by a new one (`sfWidgetFormInputFileEditable`):

Listing 6-28

```
public function configure()
{
  $this->useFields(array('filename', 'caption'));

  $this->setValidator('filename', new sfValidatorFile(array(
    'mime_types' => 'web_images',
    'path' => sfConfig::get('sf_upload_dir').'/products',
    'required' => false,
  )));

  $this->setWidget('filename', new sfWidgetFormInputFileEditable(array(
    'file_src'    => '/uploads/products/'.$this->getObject()->filename,
    'edit_mode'   => !$this->isNew(),
    'is_image'    => true,
    'with_delete' => false,
  )));

  $this->validatorSchema['caption']->setOption('required', false);
}
```

# Form Events

New to symfony 1.3 are form events that can be used to extend any form object from anywhere in the project. Symfony exposes the following four form events:

- `form.post_configure`: This event is notified after every form is configured
- `form.filter_values`: This event filters the merged, tainted parameters and files arrays just prior to binding
- `form.validation_error`: This event is notified whenever form validation fails
- `form.method_not_found`: This event is notified whenever an unknown method is called

## Custom Logging via `form.validation_error`

Using the form events, it's possible to add custom logging for validation errors on any form in your project. This might be useful if you want to track which forms and fields are causing confusion for your users.

Begin by registering a listener with the event dispatcher for the `form.validation_error` event. Add the following to the `setup()` method of `ProjectConfiguration`, which is located inside the `config` directory:

Listing 6-29

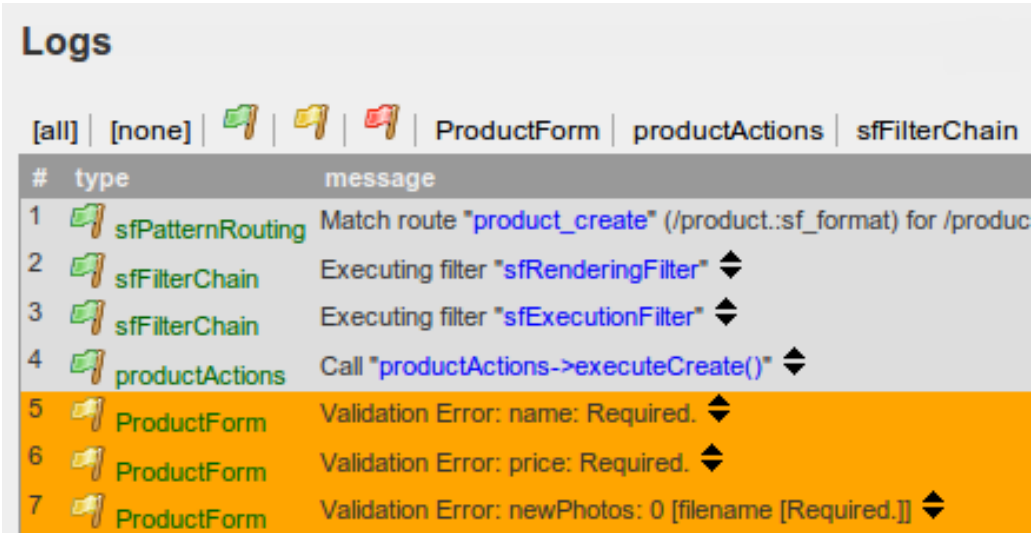
```
public function setup()
{
```

```
  // ...

  $this->getEventDispatcher()->connect(
    'form.validation_error',
    array('BaseForm', 'listenToValidationError')
  );
}
```

`BaseForm`, located in `lib/form`, is a special form class that all form classes extend. Essentially, `BaseForm` is a class where code can be placed and accessed by all form objects across the project. To enable logging of validation errors, simply add the following to the `BaseForm` class:

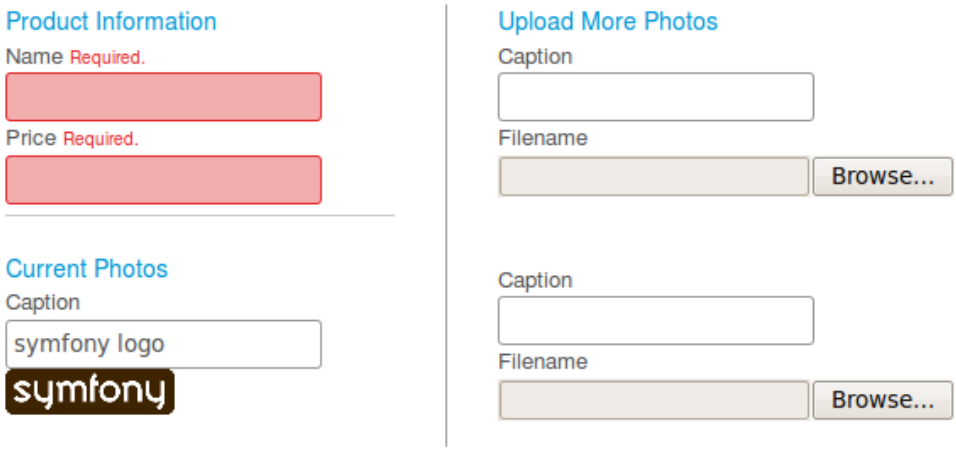
*Listing 6-30*

```
public static function listenToValidationError($event)
{
  foreach ($event['error'] as $key => $error)
  {
    self::getEventDispatcher()->notify(new sfEvent(
      $event->getSubject(),
      'application.log',
      array (
        'priority' => sfLogger::NOTICE,
        sprintf('Validation Error: %s: %s', $key, (string) $error)
      )
    ));
  }
}
```




# Custom Styling when a Form Element has an Error

As a final exercise, let's turn to a slightly lighter topic related to the styling of form elements. Suppose, for example, that the design for the `Product` page includes special styling for fields that have failed validation.




Suppose your designer has already implemented the stylesheet that will apply the error styling to any `input` field inside a `div` with the class `form_error_row`. How can we easily add the `form_row_error` class to the fields with errors?

The answer lies in a special object called a *form schema formatter*. Every symfony form uses a *form schema formatter* to determine the exact html formatting to use when outputting the form elements. By default, symfony uses a form formatter that employs HTML table tags.

First, let's create a new form schema formatter class that employs slightly lighter markup when outputting the form. Create a new file named `sfWidgetFormSchemaFormatterAc2009.class.php` and place it in the `lib/widget/` directory (you'll need to create this directory):

*Listing 6-31*

```
class sfWidgetFormSchemaFormatterAc2009 extends sfWidgetFormSchemaFormatter
{
  protected
    $rowFormat       = "<div class="form_row">
                        %label% \n %error% <br/> %field%
                        %help% %hidden_fields%\n</div>\n",
    $errorRowFormat  = "<div>%errors%</div>",
    $helpFormat      = '<div class="form_help">%help%</div>',
    $decoratorFormat = "<div>\n  %content%</div>";
}
```

Though the format of this class is strange, the general idea is that the `renderRow()` method will use the `$rowFormat` markup to organize its output. A form schema formatter class offers many other formatting options which I won't cover here in detail. For more information, consult the symfony 1.3 API[58].

To use the new form schema formatter across all form objects in your project, add the following to `ProjectConfiguration`:

*Listing 6-32*

```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    // ...

    sfWidgetFormSchema::setDefaultFormFormatterName('ac2009');
  }
}
```

58. http://www.symfony-project.org/api/1_3/sfWidgetFormSchemaFormatter

The goal is to add a `form_row_error` class to the `form_row` div element only if a field has failed validation. Add a `%row_class%` token to the `$rowFormat` property and override the `sfWidgetFormSchemaFormatter::formatRow()` method as follows:

*Listing 6-33*

```
class sfWidgetFormSchemaFormatterAc2009 extends sfWidgetFormSchemaFormatter
{
  protected
    $rowFormat       = "<div class="form_row%row_class%">
                          %label% \n %error% <br/> %field%
                          %help% %hidden_fields%\n</div>\n",
    // ...

  public function formatRow($label, $field, $errors = array(), $help = '',
$hiddenFields = null)
  {
    $row = parent::formatRow(
      $label,
      $field,
      $errors,
      $help,
      $hiddenFields
    );

    return strtr($row, array(
      '%row_class%' => (count($errors) > 0) ? ' form_row_error' : '',
    ));
  }
}
```

With this addition, each element that is output via the `renderRow()` method will automatically be surrounded by a `form_row_error` `div` if the field has failed validation.

# Final Thoughts

The form framework is simultaneously one of the most powerful and most complex components inside symfony. The trade-off for tight form validation, CSRF protection, and object forms is that extending the framework can quickly become a daunting task. Gaining a deeper understanding of the form system, however, is the key toward unlocking its potential. I hope this chapter has taken you one step closer.

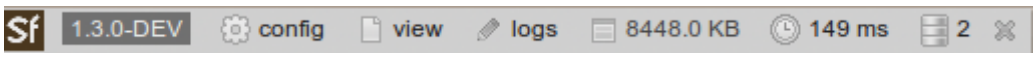
Future development of the form framework will focus on preserving the power while decreasing complexity and giving more flexibility to the developer. The form framework is only now in its infancy.

Chapter 7

# Extending the Web Debug Toolbar

*by Ryan Weaver*

By default, symfony's web debug toolbar contains a variety of tools that assist with debugging, performance enhancement and more. The web debug toolbar consists of several tools, called *web debug panels*, that relate to the cache, config, logging, memory use, symfony version, and processing time. Additionally, symfony 1.3 introduces two additional *web debug panels* for `view` information and `mail` debugging.




As of symfony 1.2, developers can easily create their own *web debug panels* and add them to the web debug toolbar. In this chapter we'll setup a new *web debug panel* and then play with all the different tools and customizations available. Additionally, the ac2009WebDebugPlugin[59] contains several useful and interesting debug panels that employ some of the techniques used in this chapter.

## Creating a New Web Debug Panel

The individual components of the web debug toolbar are known as *web debug panels* and are special classes that extend the `sfWebDebugPanel` class. Creating a new panel is actually quite easy. Create a file named `sfWebDebugPanelDocumentation.class.php` in your project's `lib/debug/` directory (you'll need to create this directory):

*Listing 7-1*

```
// lib/debug/sfWebDebugPanelDocumentation.class.php
class acWebDebugPanelDocumentation extends sfWebDebugPanel
{
  public function getTitle()
  {
    return '<img src="http://www.symfony-project.org/images/
documentation.png" alt="Documentation Shortcuts" height="16" width="16" />
docs';
  }

  public function getPanelTitle()
  {
    return 'Documentation';
  }

  public function getPanelContent()
  {
```

59. http://www.symfony-project.org/plugins/ac2009WebDebugPlugin

```
    $content = 'Placeholder Panel Content';

    return $content;
  }
}
```

At the very least, all debug panels must implement the `getTitle()`, `getPanelTitle()` and `getPanelContent()` methods.

- `sfWebDebugPanel::getTitle()`: Determines how the panel will appear in the toolbar itself. Like most panels, our custom panel includes a small icon and a short name for the panel.
- `sfWebDebugPanel::getPanelTitle()`: Used as the text for the `h1` tag that will appear at the top of the panel content. This is also used as the `title` attribute of the link tag that wraps the icon in the toolbar and as such, should *not* include any html code.
- `sfWebDebugPanel::getPanelContent()`: Generates the raw html content that will be displayed when you click on the panel icon.

The only remaining step is to notify the application that you want to include the new panel on your toolbar. To accomplish this, add a listener to the `debug.web.load_panels` event, which is notified when the web debug toolbar is collecting the potential panels. First, modify the `config/ProjectConfiguration.class.php` file to listen for the event:

*Listing 7-2*

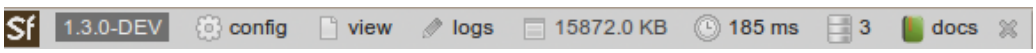
```
// config/ProjectConfiguration.class.php
public function setup()
{
  // ...

  $this->dispatcher->connect('debug.web.load_panels', array(
    'acWebDebugPanelDocumentation',
    'listenToLoadDebugWebPanelEvent'
  ));
}
```

Now, let's add the `listenToLoadDebugWebPanelEvent()` listener function to `acWebDebugPanelDocumentation.class.php` in order to add the panel to the toolbar:

*Listing 7-3*

```
// lib/debug/sfWebDebugPanelDocumentation.class.php
public static function listenToLoadDebugWebPanelEvent(sfEvent $event)
{
  $event->getSubject()->setPanel(
    'documentation',
    new self($event->getSubject())
  );
}
```

That's it! Refresh your browser and you'll instantly see the result.




---

As of symfony 1.3, a `sfWebDebugPanel` url parameter can be used to automatically open a particular web debug panel on page load. For example, adding `?sfWebDebugPanel=documentation` to the end of the url would automatically open the documentation panel we just added. This can become quite handy while building custom panels.

---

# The Three Types of Web Debug Panels

Behind the scenes, there are really three different types of web debug panels.

## The *Icon-Only* Panel Type

The most basic type of panel is one that shows an icon and text on the toolbar and nothing else. The classic example is the `memory` panel, which displays the memory use but does nothing when clicked on. To create an *icon-only* panel, simply set your `getPanelContent()` to return an empty string. The only output of the panel comes from the `getTitle()` method:

*Listing 7-4*

```
public function getTitle()
{
  $totalMemory = sprintf('%.1f', (memory_get_peak_usage(true) / 1024));

  return '<img src="'.$this->webDebug->getOption('image_root_path').'/
memory.png" alt="Memory" /> '.$totalMemory.' KB';
}

public function getPanelContent()
{
  return;
}
```

## The *Link* Panel Type

Like the *icon-only* panel, a *link* panel consists of no panel content. Unlike the *icon-only* panel, however, clicking on a *link* panel on the toolbar will take you to a url specified via the `getTitleUrl()` method of the panel. To create a *link* panel, set `getPanelContent()` to return an empty string and add a `getTitleUrl()` method to the class.

*Listing 7-5*

```
public function getTitleUrl()
{
  // link to an external uri
  return 'http://www.symfony-project.org/api/1_3/';

  // or link to a route in your application
  return url_for('homepage');
}

public function getPanelContent()
{
  return;
}
```

## The *Content* Panel Type

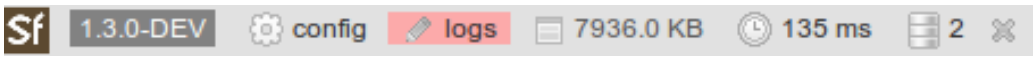
By far, the most common type of panel is a *content* panel. These panels have a full body of html content that is displayed when you click on the panel in the debug toolbar. To create this type of panel, simply make sure that the `getPanelContent()` returns more than an empty string.

# Customizing Panel Content

Now that you've created and added your custom web debug panel to the toolbar, adding content to it can be done easily via the `getPanelContent()` method. Symfony supplies several methods to assist you in making this content rich and usable.

### `sfWebDebugPanel::setStatus()`

By default, each panel on the web debug toolbar displays using the default gray background. This can be changed, however, to an orange or red background if special attention needs to be called to some content inside the panel.




To change the background color of the panel, simply employ the `setStatus()` method. This method accepts any `priority` constant from the sfLogger[60] class. In particular, there are three different status levels that correspond to the three different background colors for a panel (gray, orange and red). Most commonly, the `setStatus()` method will be called from inside the `getPanelContent()` method when some condition has occurred that needs special attention.

*Listing 7-6*
```
public function getPanelContent()
{
  // ...

  // set the background to gray (the default)
  $this->setStatus(sfLogger::INFO);

  // set the background to orange
  $this->setStatus(sfLogger::WARNING);

  // set the background to red
  $this->setStatus(sfLogger::ERR);
}
```

### `sfWebDebugPanel::getToggler()`

One of the most common features across existing web debug panels is a toggler: a visual arrow element that hides/shows a container of content when clicked.


This functionality can be easily used in the custom web debug panel via the `getToggler()` function. For example, suppose we want to toggle a list of content in a panel:

*Listing 7-7*
```
public function getPanelContent()
{
  $listContent = '<ul id="debug_documentation_list" style="display: none;">
    <li>List Item 1</li>
    <li>List Item 2</li>
```

60. http://www.symfony-project.org/api/1_3/sfLogger

```
  </ul>';

  $toggler = $this->getToggler('debug_documentation_list', 'Toggle list');

  return sprintf('<h3>List Items %s</h3>%s',  $toggler, $listContent);
}
```

The `getToggler` takes two arguments: the DOM `id` of the element to toggle and a `title` to set as the `title` attribute of the toggler link. It's up to you to create the DOM element with the given `id` attribute as well as any descriptive label (e.g. "List Items") for the toggler.

### `sfWebDebugPanel::getToggleableDebugStack()`

Similar to `getToggler()`, `getToggleableDebugStack()` renders a clickable arrow that toggles the display of a set of content. In this case, the set of content is a debug stack trace. This function is useful if you need to display log results for a custom class. For example, suppose we perform some custom logging on a class called `myCustomClass`:

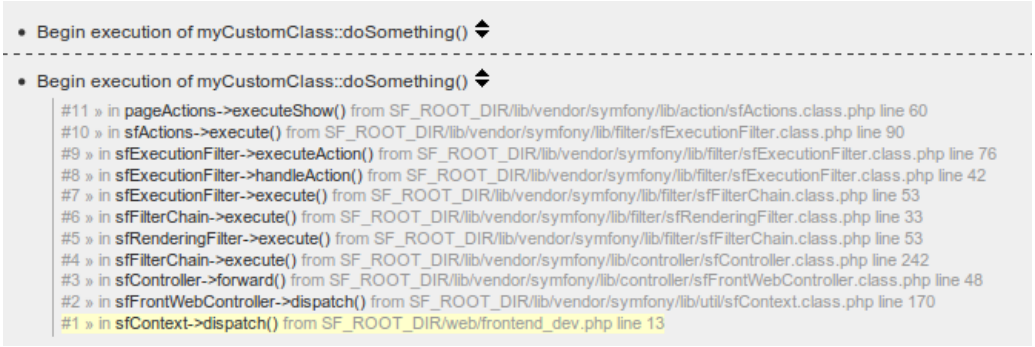
*Listing 7-8*

```
class myCustomClass
{
  public function doSomething()
  {
    $dispatcher = sfApplicationConfiguration::getActive()
      ->getEventDispatcher();

    $dispatcher->notify(new sfEvent($this, 'application.log', array(
      'priority' => sfLogger::INFO,
      'Beginning execution of myCustomClass::doSomething()',
    )));
  }
}
```

As an example, let's display a list of the log messages related to `myCustomClass` complete with debug stack traces for each.

*Listing 7-9*

```
public function getPanelContent()
{
  // retrieves all of the log messages for the current request
  $logs = $this->webDebug->getLogger()->getLogs();

  $logList = '';
  foreach ($logs as $log)
  {
    if ($log['type'] == 'myCustomClass')
    {
      $logList .= sprintf('<li>%s %s</li>',
        $log['message'],
        $this->getToggleableDebugStack($log['debug_backtrace'])
      );
    }
  }

  return sprintf('<ul>%s</ul>', $logList);
}
```




Even without creating a custom panel, the log messages for `myCustomClass` would be displayed on the logs panel. The advantage here is simply to collect this subset of log messages in one location and control its output.

### `sfWebDebugPanel::formatFileLink()`

New to symfony 1.3 is the ability to click on files in the web debug toolbar and have them open in your preferred text editor. For more information, see the "What's new"[61] article for symfony 1.3.

To activate this feature for any particular file path, the `formatFileLink()` must be used. In addition to the file itself, an exact line can optionally be targeted. For example, the following code would link to line 15 of `config/ProjectConfiguration.class.php`:

*Listing 7-10*

```
public function getPanelContent()
{
  $content = '';

  // ...

  $path = sfConfig::get('sf_config_dir') .
'/ProjectConfiguration.class.php';
  $content .= $this->formatFileLink($path, 15, 'Project Configuration');

  return $content;
}
```

Both the second argument (the line number) and the third argument (the link text) are optional. If no "link text" argument is specified, the file path will be shown as the text of the link.

Before testing, be sure you've configured the new file linking feature. This feature can be setup via the `sf_file_link_format` key in `settings.yml` or via the `file_link_format` setting in xdebug[62]. The latter method ensures that the project isn't bound to a specific IDE.

61. http://www.symfony-project.org/tutorial/1_3/en/whats-new
62. http://xdebug.org/docs/stack_trace#file_link_format

# Other Tricks with the Web Debug Toolbar

For the most part, the magic of your custom web debug panel will be contained in the content and information you choose to display. There are, however, a few more tricks worth exploring.

## Removing Default Panels

By default, symfony automatically loads several web debug panels into your web debug toolbar. By using the `debug.web.load_panels` event, these default panels can also be easily removed. Use the same listener function declared earlier, but replace the body with the `removePanel()` function. The following code will remove the `memory` panel from the toolbar:

*Listing 7-11*

```
public static function listenToLoadDebugWebPanelEvent(sfEvent $event)
{
  $event->getSubject()->removePanel('memory');
}
```

## Accessing the Request Parameters from a Panel

One of the most common things needed inside a web debug panel is the request parameters. Say, for example, that you want to display information from the database about an `Event` object in the database based off of an `event_id` request parameter:

*Listing 7-12*

```
$parameters = $this->webDebug->getOption('request_parameters');
if (isset($parameters['event_id']))
{
  $event = Doctrine::getTable('Event')->find($parameters['event_id']);
}
```

## Conditionally Hide a Panel

Sometimes, your panel may not have any useful information to display for the current request. In these situations, you can choose to hide your panel altogether. Let's suppose, in the previous example, that the custom panel displays no information unless an `event_id` request parameter is present. To hide the panel, simply return no content from the `getTitle()` method:

*Listing 7-13*

```
public function getTitle()
{
  $parameters = $this->webDebug->getOption('request_parameters');
  if (!isset($parameters['event_id']))
  {
    return;
  }

  return '<img src="http://www.symfony-project.org/acWebDebugPlugin/images/
documentation.png" alt="Documentation Shortcuts" height="16" width="16" />
docs';
}
```

## Final Thoughts

The web debug toolbar exists to make the developer's life easier, but it's more than a passive display of information. By adding custom web debug panels, the potential of the web debug toolbar is limited only by the imagination of the developers. The ac2009WebDebugPlugin[63] includes only some of the panels that could be created. Feel free to create your own.

63. http://www.symfony-project.org/plugins/ac2009WebDebugPlugin

Chapter 8

# Advanced Doctrine Usage

*By Jonathan H. Wage*

## Writing a Doctrine Behavior

In this section we will demonstrate how you can write a behavior using Doctrine 1.2. We'll create an example that allows you to easily maintain a cached count of relationships so that you don't have to query the count every single time.

The functionality is quite simple. For all the relationships you want to maintain a count for, the behavior will add a column to the model to store the current count.

### The Schema

Here is the schema we will use to start with. Later we will modify this and add the `actAs` definition for the behavior we are about to write:

*Listing 8-1*

```
# config/doctrine/schema.yml
Thread:
  columns:
    title:
      type: string(255)
      notnull: true

Post:
  columns:
    thread_id:
      type: integer
      notnull: true
    body:
      type: clob
      notnull: true
  relations:
    Thread:
      onDelete: CASCADE
      foreignAlias: Posts
```

Now we can build everything for that schema:

*Listing 8-2*

```
$ php symfony doctrine:build --all
```

## The Template

First we need to write the basic `Doctrine_Template` child class that will be responsible for adding the columns to the model that will store the counts.

You can simply put this somewhere in one of the project `lib/` directories and symfony will be able to autoload it for you:

Listing 8-3
```
// lib/count_cache/CountCache.class.php
class CountCache extends Doctrine_Template
{
  public function setTableDefinition()
  {
  }

  public function setUp()
  {
  }
}
```

Now let's modify the `Post` model to `actAs` the `CountCache` behavior:

Listing 8-4
```
# config/doctrine/schema.yml
Post:
  actAs:
    CountCache: ~
  # ...
```

Now that we have the `Post` model using the `CountCache` behavior let me explain a little about what happens with it.

When the mapping information for a model is instantiated, any attached behaviors get the `setTableDefinition()` and `setUp()` methods invoked. Just like you have in the `BasePost` class in `lib/model/doctrine/base/BasePost.class.php`. This allows you to add things to any model in a plug n' play fashion. This can be columns, relationships, event listeners, etc.

Now that you understand a little bit about what is happening, let's make the `CountCache` behavior actually do something:

Listing 8-5
```
class CountCache extends Doctrine_Template
{
  protected $_options = array(
    'relations' => array()
  );

  public function setTableDefinition()
  {
    foreach ($this->_options['relations'] as $relation => $options)
    {
      // Build column name if one is not given
      if (!isset($options['columnName']))
      {
        $this->_options['relations'][$relation]['columnName'] =
'num_'.Doctrine_Inflector::tableize($relation);
      }

      // Add the column to the related model
      $columnName = $this->_options['relations'][$relation]['columnName'];
```

```
      $relatedTable = $this->_table->getRelation($relation)->getTable();
      $this->_options['relations'][$relation]['className'] =
$relatedTable->getOption('name');
      $relatedTable->setColumn($columnName, 'integer', null,
array('default' => 0));
    }
  }
}
```

The above code will now add columns to maintain the count on the related model. So in our case, we're adding the behavior to the `Post` model for the `Thread` relationship. We want to maintain the number of posts any given `Thread` instance has in a column named `num_posts`. So now modify the YAML schema to define the extra options for the behavior:

*Listing 8-6*

```
# ...

Post:
  actAs:
    CountCache:
      relations:
        Thread:
          columnName: num_posts
          foreignAlias: Posts
  # ...
```

Now the `Thread` model has a `num_posts` column that we will keep up to date with the number of posts that each thread has.

## The Event Listener

The next step to building the behavior is to write a record event listener that will be responsible for keeping the count up to date when we insert new records, delete a record or batch DQL delete records:

*Listing 8-7*

```
class CountCache extends Doctrine_Template
{
  // ...

  public function setTableDefinition()
  {
    // ...

    $this->addListener(new CountCacheListener($this->_options));
  }
}
```

Before we go any further we need to define the `CountCacheListener` class that extends `Doctrine_Record_Listener`. It accepts an array of options that are simply forwarded to the listener from the template:

*Listing 8-8*

```
// lib/model/count_cache/CountCacheListener.class.php

class CountCacheListener extends Doctrine_Record_Listener
{
  protected $_options;

  public function __construct(array $options)
```

```
  {
    $this->_options = $options;
  }
}
```

Now we must utilize the following events in order to keep our counts up to date:

- **postInsert()**: Increments the count when a new object is inserted;
- **postDelete()**: Decrements the count when a object is deleted;
- **preDqlDelete()**: Decrements the counts when records are deleted through a DQL delete.

First let's define the `postInsert()` method:

Listing 8-9
```
class CountCacheListener extends Doctrine_Record_Listener
{
  // ...

  public function postInsert(Doctrine_Event $event)
  {
    $invoker = $event->getInvoker();
    foreach ($this->_options['relations'] as $relation => $options)
    {
      $table = Doctrine::getTable($options['className']);
      $relation = $table->getRelation($options['foreignAlias']);

      $table
        ->createQuery()
        ->update()
        ->set($options['columnName'], $options['columnName'].' + 1')
        ->where($relation['local'].' = ?', $invoker->$relation['foreign'])
        ->execute();
    }
  }
}
```

The above code will increment the counts by one for all the configured relationships by issuing a DQL UPDATE query when a new object like below is inserted:

Listing 8-10
```
$post = new Post();
$post->thread_id = 1;
$post->body = 'body of the post';
$post->save();
```

The `Thread` with an `id` of `1` will get the `num_posts` column incremented by `1`.

Now that the counts are being incremented when new objects are inserted, we need to handle when objects are deleted and decrement the counts. We will do this by implementing the `postDelete()` method:

Listing 8-11
```
class CountCacheListener extends Doctrine_Record_Listener
{
  // ...

  public function postDelete(Doctrine_Event $event)
  {
    $invoker = $event->getInvoker();
    foreach ($this->_options['relations'] as $relation => $options)
```

```
    {
      $table = Doctrine::getTable($options['className']);
      $relation = $table->getRelation($options['foreignAlias']);

      $table
        ->createQuery()
        ->update()
        ->set($options['columnName'], $options['columnName'].' - 1')
        ->where($relation['local'].' = ?', $invoker->$relation['foreign'])
        ->execute();
    }
  }
}
```

The above `postDelete()` method is almost identical to the `postInsert()` the only difference is we decrement the `num_posts` column by `1` instead of incrementing it. It handles the following code if we were to delete the `$post` record we saved previously:

```
$post->delete();
```

*Listing 8-12*

The last piece to the puzzle is to handle when records are deleted using a DQL update. We can solve this by using the `preDqlDelete()` method:

```
class CountCacheListener extends Doctrine_Record_Listener
{
  // ...

  public function preDqlDelete(Doctrine_Event $event)
  {
    foreach ($this->_options['relations'] as $relation => $options)
    {
      $table = Doctrine::getTable($options['className']);
      $relation = $table->getRelation($options['foreignAlias']);

      $q = clone $event->getQuery();
      $q->select($relation['foreign']);
      $ids = $q->execute(array(), Doctrine::HYDRATE_NONE);

      foreach ($ids as $id)
      {
        $id = $id[0];

        $table
          ->createQuery()
          ->update()
          ->set($options['columnName'], $options['columnName'].' - 1')
          ->where($relation['local'].' = ?', $id)
          ->execute();
      }
    }
  }
}
```

*Listing 8-13*

The above code clones the `DQL DELETE` query and transforms it to a `SELECT` which allows us to retrieve the `ID`s that will be deleted so that we can update the counts of those records that were deleted.

Now we have the following scenario taken care of and the counts will be decremented if we were to do the following:

Listing 8-14
```
Doctrine::getTable('Post')
  ->createQuery()
  ->delete()
  ->where('id = ?', 1)
  ->execute();
```

Or even if we were to delete multiple records the counts would still be decremented properly:

Listing 8-15
```
Doctrine::getTable('Post')
  ->createQuery()
  ->delete()
  ->where('body LIKE ?', '%cool%')
  ->execute();
```

---

In order for the `preDqlDelete()` method to be invoked you must enable an attribute. The DQL callbacks are off by default due to them costing a little extra. So if you want to use them, you must enable them.

Listing 8-16
```
$manager->setAttribute(Doctrine_Core::ATTR_USE_DQL_CALLBACKS, true);
```

---

That's it! The behavior is finished. The last thing we'll do is test it out a bit.

## Testing

Now that we have the code implemented, let's give it a test run with some sample data fixtures:

Listing 8-17
```
# data/fixtures/data.yml

Thread:
  thread1:
    title: Test Thread
    Posts:
      post1:
        body: This is the body of my test thread
      post2:
        body: This is really cool
      post3:
        body: Ya it is pretty cool
```

Now, build everything again and load the data fixtures:

Listing 8-18
```
$ php symfony doctrine:build --all --and-load
```

Now everything is built and the data fixtures are loaded; so let's run a test to see if the counts have been kept up to date:

Listing 8-19
```
$ php symfony doctrine:dql "FROM Thread t, t.Posts p"
doctrine - executing: "FROM Thread t, t.Posts p" ()
doctrine -   id: '1'
doctrine -   title: 'Test Thread'
doctrine -   num_posts: '3'
```

```
doctrine -   Posts:
doctrine -     -
doctrine -       id: '1'
doctrine -       thread_id: '1'
doctrine -       body: 'This is the body of my test thread'
doctrine -     -
doctrine -       id: '2'
doctrine -       thread_id: '1'
doctrine -       body: 'This is really cool'
doctrine -     -
doctrine -       id: '3'
doctrine -       thread_id: '1'
doctrine -       body: 'Ya it is pretty cool'
```

You will see the `Thread` model has a `num_posts` column, whose value is three. If we were to delete one of the posts with the following code it will decrement the count for you:

*Listing 8-20*

```
$post = Doctrine_Core::getTable('Post')->find(1);
$post->delete();
```

You will see that the record is deleted and the count is updated:

*Listing 8-21*

```
$ php symfony doctrine:dql "FROM Thread t, t.Posts p"
doctrine - executing: "FROM Thread t, t.Posts p" ()
doctrine -   id: '1'
doctrine -   title: 'Test Thread'
doctrine -   num_posts: '2'
doctrine -   Posts:
doctrine -     -
doctrine -       id: '2'
doctrine -       thread_id: '1'
doctrine -       body: 'This is really cool'
doctrine -     -
doctrine -       id: '3'
doctrine -       thread_id: '1'
doctrine -       body: 'Ya it is pretty cool'
```

This even works if we were to batch delete the remaining two records with a DQL delete query:

*Listing 8-22*

```
Doctrine_Core::getTable('Post')
  ->createQuery()
  ->delete()
  ->where('body LIKE ?', '%cool%')
  ->execute();
```

Now we've deleted all the related posts and the `num_posts` should be zero:

*Listing 8-23*

```
$ php symfony doctrine:dql "FROM Thread t, t.Posts p"
doctrine - executing: "FROM Thread t, t.Posts p" ()
doctrine -   id: '1'
doctrine -   title: 'Test Thread'
doctrine -   num_posts: '0'
doctrine -   Posts: {  }
```

That is it! I hope this article was both useful in the sense that you learned something about behaviors and the behavior itself is also useful to you!

# Using Doctrine Result Caching

In heavily trafficked web applications it is a common need to cache information to save your CPU resources. With the latest Doctrine 1.2 we have made a lot of improvements to the result set caching that gives you much more control over deleting cache entries from the cache drivers. Previously it was not possible to specify the cache key used to store the cache entry so you couldn't really identify the cache entry in order to delete it.

In this section we will show you a simple example of how you can utilize result set caching to cache all your user related queries as well as using events to make sure they are properly cleared when some data is changed.

## Our Schema

For this example, let's use the following schema:

Listing 8-24
```
# config/doctrine/schema.yml
User:
  columns:
    username:
      type: string(255)
      notnull: true
      unique: true
    password:
      type: string(255)
      notnull: true
```

Now let's build everything from that schema with the following command:

Listing 8-25
```
$ php symfony doctrine:build --all
```

Once you do that, you should have the following `User` class generated for you:

Listing 8-26
```
// lib/model/doctrine/User.class.php
/**
 * User
 *
 * This class has been auto-generated by the Doctrine ORM Framework
 *
 * @package    ##PACKAGE##
 * @subpackage ##SUBPACKAGE##
 * @author     ##NAME## <##EMAIL##>
 * @version    SVN: $Id: Builder.php 6508 2009-10-14 06:28:49Z jwage $
 */
class User extends BaseUser
{
}
```

Later in the article you will need to add some code to this class so make note of it.

## Configuring Result Cache

In order to use the result cache we need to configure a cache driver for the queries to use. This can be done by setting the `ATTR_RESULT_CACHE` attribute. We will use the APC cache driver as it is the best choice for production. If you do not have APC available, you can use the `Doctrine_Cache_Db` or `Doctrine_Cache_Array` driver for testing purposes.

We can set this attribute in our `ProjectConfiguration` class. Define a `configureDoctrine()` method:

```
// config/ProjectConfiguration.class.php
```

Listing 8-27

```
// ...
class ProjectConfiguration extends sfProjectConfiguration
{
  // ...

  public function configureDoctrine(Doctrine_Manager $manager)
  {
    $manager->setAttribute(Doctrine_Core::ATTR_RESULT_CACHE, new
Doctrine_Cache_Apc());
  }
}
```

Now that we have the result cache driver configured, we can begin to actually use this driver to cache the result sets of the queries.

## Sample Queries

Now imagine in your application you have a bunch of user related queries and you want to clear them whenever some user data is changed.

Here is a simple query that we might use to render a list of users sorted alphabetically:

```
$q = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->orderBy('u.username ASC');
```

Listing 8-28

Now, we can turn caching on for that query by using the `useResultCache()` method:

```
$q->useResultCache(true, 3600, 'users_index');
```

Listing 8-29

Notice the third argument. This is the key that will be used to store the cache entry for the results in the cache driver. This allows us to easily identify that query and delete it from the cache driver.

Now when we execute the query it will query the database for the results and store them in the cache driver under the key named `users_index` and any subsequent requests will get the information from the cache driver instead of asking the database:

```
$users = $q->execute();
```

Listing 8-30

Not only does this save processing on your database server, it also bypasses the entire hydration process as Doctrine stores the hydrated data. This means that it will relieve some processing on your web server as well.

Now if we check in the cache driver, you will see that there is an entry named `users_index`:

```
if ($cacheDriver->contains('users_index'))
{
  echo 'cache exists';
}
else
```

Listing 8-31

```
{
  echo 'cache does not exist';
}
```

## Deleting Cache

Now that the query is cached, we need to learn a bit about how we can delete that cache. We can delete it manually using the cache driver API or we can utilize some events to automatically clear the cache entry when a user is inserted or modified.

### Cache Driver API

First we will just demonstrate the raw API of the cache driver before we implement it in an event.

---

To get access to the result cache driver instance you can get it from the `Doctrine_Manager` class instance.

Listing 8-32
```
$cacheDriver = $manager->getAttribute(Doctrine_Core::ATTR_RESULT_CACHE);
```

If you don't already have access to the `$manager` variable already you can retrieve the instance with the following code.

Listing 8-33
```
$manager = Doctrine_Manager::getInstance();
```

---

Now we can begin to use the API to delete our cache entries:

Listing 8-34
```
$cacheDriver->delete('users_index');
```

You probably have more than one query prefixed with `users_` and it would make sense to delete the result cache for all of them. In this case, the `delete()` method by itself will not work. For this we have a method named `deleteByPrefix()`, which allows us to delete any cache entry that contains the given prefix. Here is an example:

Listing 8-35
```
$cacheDriver->deleteByPrefix('users_');
```

We have a few other convenient methods we can use to delete cache entries if the `deleteByPrefix()` is not sufficient for you:

- `deleteBySuffix($suffix)`: Deletes cache entries that have the passed suffix;
- `deleteByRegexp($regex)`: Deletes cache entries that match the passed regular expression;
- `deleteAll()`: Deletes all cache entries.

## Deleting with Events

The ideal way to clear the cache would be to automatically clear it whenever some user data is modified. We can do this by implementing a `postSave()` event in our `User` model class definition.

Remember the `User` class we talked about earlier? Now we need to add some code to it so open the class in your favorite editor and add the following `postSave()` method:

Listing 8-36
```
// lib/model/doctrine/User.class.php
```

```
class User extends BaseUser
{
  // ...

  public function postSave($event)
  {
    $cacheDriver =
$this->getTable()->getAttribute(Doctrine_Core::ATTR_RESULT_CACHE);
    $cacheDriver->deleteByPrefix('users_');
  }
}
```

Now if we were to update a user or insert a new user it would clear the cache for all the user related queries:

Listing 8-37

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->save();
```

The next time the queries are invoked it will see the cache does not exist and fetch the data fresh from the database and cache it again for subsequent requests.

While this example is very simple it should demonstrate pretty well how you can use these features to implement fine grained caching on your Doctrine queries.

# Writing a Doctrine Hydrator

One of the key features of Doctrine is the ability to transform a `Doctrine_Query` object to the various result set structures. This is the job of the Doctrine hydrators and up until Doctrine 1.2, the hydrators were all hardcoded and not open for developers to write custom ones. Now that this has changed it is possible to write a custom hydrator and create whatever data structure that is desired from the data the database gives you back when executing a `Doctrine_Query` instance.

In this example we will build a hydrator that will be extremely simple and easy to understand, yet very useful. It will allow you to select two columns and hydrate the data to a flat array where the first selected column is the key and the second select column is the value.

## The Schema and Fixtures

To get started first we need a simple schema to run our tests with. We will just use a simple `User` model:

Listing 8-38

```
# config/doctrine/schema.yml
User:
  columns:
    username: string(255)
    is_active: string(255)
```

We will also need some data fixtures to test against, so copy the fixtures from below:

Listing 8-39

```
# data/fixtures/data.yml
User:
  user1:
    username: jwage
```

```
    password: changeme
    is_active: 1
  user2:
    username: jonwage
    password: changeme
    is_active: 0
```

Now build everything with the following command:

Listing 8-40
```
$ php symfony doctrine:build --all --and-load
```

## Writing the Hydrator

To write a hydrator all we need to do is write a new class which extends `Doctrine_Hydrator_Abstract` and must implement a `hydrateResultSet($stmt)` method. It receives the `PDOStatement` instance used to execute the query. We can then use that statement to get the raw results of the query from PDO then transform that to our own structure.

Let's create a new class named `KeyValuePairHydrator` and place it in the `lib/` directory so that symfony can autoload it:

Listing 8-41
```
// lib/KeyValuePairHydrator.class.php
class KeyValuePairHydrator extends Doctrine_Hydrator_Abstract
{
  public function hydrateResultSet($stmt)
  {
    return $stmt->fetchAll(Doctrine_Core::FETCH_NUM);
  }
}
```

The above code as it is now would just return the data exactly as it is from PDO. This isn't quite what we want. We want to transform that data to our own key => value pair structure. So let's modify the `hydrateResultSet()` method a little bit to do what we want:

Listing 8-42
```
// lib/KeyValuePairHydrator.class.php
class KeyValuePairHydrator extends Doctrine_Hydrator_Abstract
{
  public function hydrateResultSet($stmt)
  {
    $results = $stmt->fetchAll(Doctrine_Core::FETCH_NUM);
    $array = array();
    foreach ($results as $result)
    {
      $array[$result[0]] = $result[1];
    }

    return $array;
  }
}
```

Well that was easy! The hydrator code is finished and it does exactly what we want so let's test it!

## Using the Hydrator

To use and test the hydrator we first need to register it with Doctrine so that when we execute some queries, Doctrine is aware of the hydrator class we have written.

To do this, register it on the `Doctrine_Manager` instance in the `ProjectConfiguration`:

*Listing 8-43*

```
// config/ProjectConfiguration.class.php

// ...
class ProjectConfiguration extends sfProjectConfiguration
{
  // ...

  public function configureDoctrine(Doctrine_Manager $manager)
  {
    $manager->registerHydrator('key_value_pair', 'KeyValuePairHydrator');
  }
}
```

Now that we have the hydrator registered, we can make use of it with `Doctrine_Query` instances. Here is an example:

*Listing 8-44*

```
$q = Doctrine_Core::getTable('User')
  ->createQuery('u')
  ->select('u.username, u.is_active');

$results = $q->execute(array(), 'key_value_pair');
print_r($results);
```

Executing the above query with the data fixtures we defined above would result in the following:

*Listing 8-45*

```
Array
(
    [jwage] => 1
    [jonwage] => 0
)
```

Well that is it! Pretty simple huh? I hope this will be useful for you and as a result the community gets some awesome new hydrators contributed.

Chapter 9

# Taking Advantage of Doctrine Table Inheritance

*by Hugo Hamon*

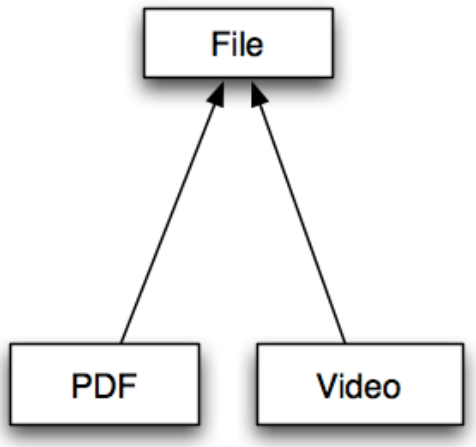
As of symfony 1.3 Doctrine has officially become the default ORM library while Propel's development has slowed down over the last few months. The Propel project is still supported and continues to be improved thanks to the efforts of symfony community members.

The Doctrine 1.2 project became the new default symfony ORM library both because it is easier to use than Propel and because it bundles a lot of great features including behaviors, easy DQL queries, migrations and table inheritance.

This chapter describes what table inheritance is and how it is now fully integrated in symfony 1.3. Thanks to a real-world example, this chapter will illustrate how to leverage Doctrine table inheritance to make code more flexible and better organized.

## Doctrine Table Inheritance

Though not really known and used by many developers, table inheritance is probably one of the most interesting features of Doctrine. Table inheritance allows the developer to create database tables that inherit from each other in the same way that classes inherit in an object oriented programming language. Table inheritance provides an easy way to share data between two or more tables in a single super table. Look at the diagram below to better understand the table inheritance principle.



Doctrine provides three different strategies to manage table inheritances depending on the application's needs (performance, atomicity, simplicity...): **simple**, **column aggregation** and **concrete** table inheritance. While all of these strategies are described in the Doctrine book[64], some further explanation will help to better understand each option and in which circumstances they are useful.

## The Simple Table Inheritance Strategy

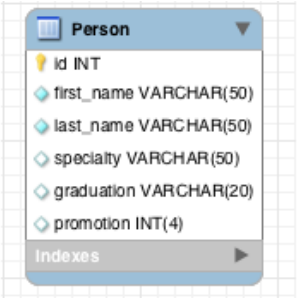
The simple table inheritance strategy is the simplest of all as it stores all columns, including children tables columns, in the super parent table. If the model schema looks like the following YAML code, Doctrine will generate one single table `Person`, which includes both the `Professor` and `Student` tables' columns.

*Listing 9-1*

```
Person:
  columns:
    first_name:
      type:           string(50)
      notnull:        true
    last_name:
      type:           string(50)
      notnull:        true

Professor:
  inheritance:
    type:             simple
    extends:          Person
  columns:
    specialty:
      type:           string(50)
      notnull:        true

Student:
  inheritance:
    type:             simple
    extends:          Person
  columns:
    graduation:
      type:           string(20)
      notnull:        true
    promotion:
      type:           integer(4)
      notnull:        true
```

With the simple inheritance strategy, the extra columns `specialty`, `graduation` and `promotion` are automatically registered at the top level in the `Person` model even if Doctrine generates one model class for both `Student` and `Professor` tables.

64. http://www.doctrine-project.org/documentation/1_2/en




This strategy has an important drawback as the super parent table `Person` does not provide any column to identify each record's type. In other words, there is no way to retrieve only `Professor` or `Student` objects. The following Doctrine statement returns a `Doctrine_Collection` of all table records (`Student` and `Professor` records).

*Listing 9-2*

```
$professors = Doctrine_Core::getTable('Professor')->findAll();
```

The simple table inheritance strategy is not really useful in real world examples as there is generally the need to select and hydrate objects of a specific type. Consequently, it won't be used further in this chapter.

## The Column Aggregation Table Inheritance Strategy

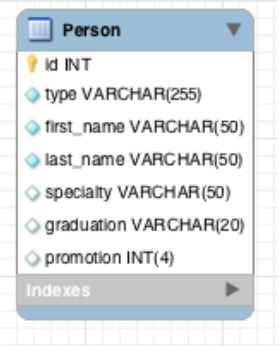
The column aggregation table inheritance strategy is similar to the simple inheritance strategy except that it includes a `type` column to identify the different record types. Consequently, when a record is persisted to the database, a type value is added to it in order to store the class to which it belongs.

*Listing 9-3*

```
Person:
  columns:
    first_name:
      type:           string(50)
      notnull:        true
    last_name:
      type:           string(50)
      notnull:        true

Professor:
  inheritance:
    type:             column_aggregation
    extends:          Person
    keyField:         type
    keyValue:         1
  columns:
    specialty:
      type:           string(50)
      notnull:        true

Student:
  inheritance:
    type:             column_aggregation
    extends:          Person
    keyField:         type
    keyValue:         2
  columns:
    graduation:
      type:           string(20)
```

```
      notnull:        true
    promotion:
      type:           integer(4)
      notnull:        true
```

In the above YAML schema, the inheritance type has been changed to `column_aggregation` and two new attributes have been added. The first attribute, `keyField`, specifies the column that will be created to store the type information for each record. The `keyField` is a string column named `type`, which is the default column name if no `keyField` is specified. The second attribute defines the type value for each record that belong to the `Professor` or `Student` classes.




The column aggregation strategy is a good method for table inheritance as it creates one single table (`Person`) containing all defined fields plus the `type` field. Consequently, there is no need to make several tables and join them with an SQL query. Below are some examples of how to query tables and which type of results will be returned:

*Listing 9-4*

```
// Returns a Doctrine_Collection of Professor objects
$professors = Doctrine_Core::getTable('Professor')->findAll();

// Returns a Doctrine_Collection of Student objects
$students = Doctrine_Core::getTable('Student')->findAll();

// Returns a Professor object
$professor =
Doctrine_Core::getTable('Professor')->findOneBySpeciality('physics');

// Returns a Student object
$student = Doctrine_Core::getTable('Student')->find(42);

// Returns a Student object
$student = Doctrine_Core::getTable('Person')->findOneByIdAndType(array(42,
2));
```

When performing data retrieval from a subclass (`Professor`, `Student`), Doctrine will automatically append the SQL `WHERE` clause to the query on the `type` column with the corresponding value.

However, there are some drawbacks to using the column aggregation strategy in certain cases. First, column aggregation prevents each sub-table's fields from being set as `required`. Depending on how many fields there are, the `Person` table may contain records with several empty values.

The second drawback relates to the number of sub-tables and fields. If the schema declares a lot of sub-tables, which in turn declare a lot of fields, the final super table will consist of a very large number of columns. Consequently, the table may be more difficult to maintain.

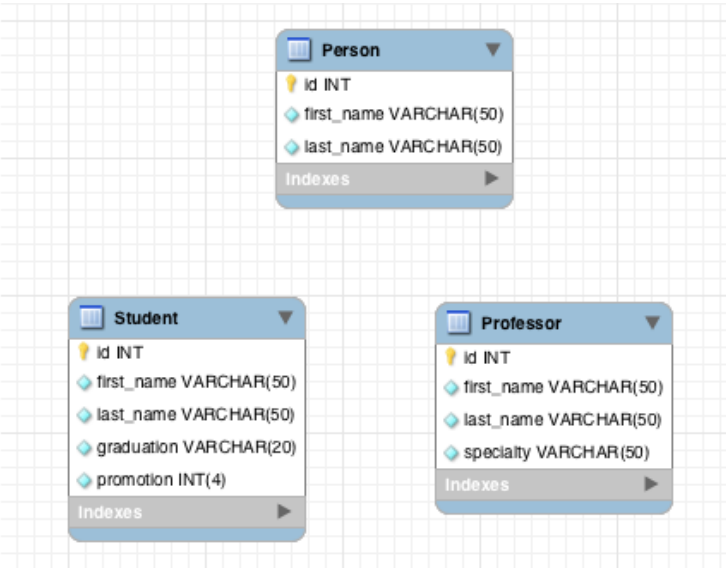
## The Concrete Table Inheritance Strategy

The concrete table inheritance strategy is a good compromise between the advantages of the column aggregation strategy, performance and maintainability. Indeed, this strategy creates independent tables for each subclass containing all columns: both the shared columns and the model's independent columns.

*Listing 9-5*

```
Person:
  columns:
    first_name:
      type:           string(50)
      notnull:        true
    last_name:
      type:           string(50)
      notnull:        true

Professor:
  inheritance:
    type:             concrete
    extends:          Person
  columns:
    specialty:
      type:           string(50)
      notnull:        true

Student:
  inheritance:
    type:             concrete
    extends:          Person
  columns:
    graduation:
      type:           string(20)
      notnull:        true
    promotion:
      type:           integer(4)
      notnull:        true
```

So, for the previous schema, the generated `Professor` table will contain the following set of fields : `id`, `first_name`, `last_name` and `specialty`.



This approach has several advantages against previous strategies. The first one is that all tables are isolated and remain independent of each other. Additionally, there are no more blank fields and the extra `type` column is not included. The result is that each table is lighter and isolated from the other tables.

The fact that shared fields are duplicated in sub-tables is a gain for performance and scalability as Doctrine does not need to make an automatic SQL join on a super table to retrieve shared data belonging to a sub-tables record.

The only two drawbacks of the concrete table inheritance strategy are the shared fields duplication (though duplication is generally the key for performance) and the fact that the generated super table will always be empty. Indeed, Doctrine has generated a `Person` table though it won't be filled or referenced by any query. No query will be performed on that table as everything is stored in subtables.

We just took the time to introduce the three Doctrine table inheritance strategies but we've not yet tried them in a real world example with symfony. The following part of this chapter explains how to take advantage of the Doctrine table inheritance in symfony 1.3, particularly within the model and the form framework.

# Symfony Integration of Table Inheritance

Before symfony 1.3, Doctrine table inheritance wasn't fully supported by the framework as form and filter classes didn't correctly inherit from the base class. Consequently, developers who needed to use table inheritance were forced to tweak forms and filters and were obliged to override lots of methods to retrieve the inheritance behavior.

Thanks to community feedback, the symfony core team has improved the forms and filters in order to easily and fully support Doctrine table inheritance in symfony 1.3.

The remainder of this chapter will explain how to use Doctrine's table inheritance and how to take advantage of it in several situations including in the models, forms, filters and admin generators. Real case study examples will help us to better understand how inheritance works with symfony so that you can easily use it for your own needs.

## Introducing the Real World Case Studies

Throughout this chapter, several real world case studies will be presented to expose the many advantages of the Doctrine table inheritance approach at several levels: in `models`, `forms`, `filters` and the `admin generator`.

The first example comes from an application developed at Sensio for a well known French company. It shows how Doctrine table inheritance is a good solution to manage a dozen identical referential sets of data in order to share methods and properties and avoid code duplication.

The second example shows how to take advantage of the concrete table inheritance strategy with forms by creating a simple model to manage digital files.

Finally, the third example will demonstrate how to take advantage of the table inheritance with the Admin Generator, and how to make it more flexible.

## Table Inheritance at the Model Layer

Similar to the Object Oriented Programming concept, table inheritance encourages data sharing. Consequently, it allows for the sharing of properties and methods when dealing with

generated models. Doctrine table inheritance is a good way to share and override actions callable on inherited objects. Let's explain this concept with a real world example.

## The Problem

Lots of web applications require "referential" data in order to function. A referential is generally a small set of data represented by a simple table containing at least two fields (e.g. `id` and `label`). In some cases, however, the referential contains extra data such as an `is_active` or `is_default` flag. This was the case recently at Sensio with a customer application.

The customer wanted to manage a large set of data, which drove the main forms and views of the application. All of these referential tables were built around the same basic model: `id`, `label`, `position` and `is_default`. The `position` field helps to rank records thanks to an ajax drag and drop functionality. The `is_default` field represents a flag that indicates whether or not a record should to be set as "selected" by default when it feeds an HTML select dropdown box.

## The Solution

Managing more than two equal tables is one of the best problems to solve with table inheritance. In the above problem, concrete table inheritance was selected to fit the needs and to share each object's methods in a single class. Let's have a look at the following simplified schema, which illustrates the problem.

*Listing 9-6*

```
sfReferential:
  columns:
    id:
      type:         integer(2)
      notnull:      true
    label:
      type:         string(45)
      notnull:      true
    position:
      type:         integer(2)
      notnull:      true
    is_default:
      type:         boolean
      notnull:      true
      default:      false

sfReferentialContractType:
  inheritance:
    type:           concrete
    extends:        sfReferential

sfReferentialProductType:
  inheritance:
    type:           concrete
    extends:        sfReferential
```

Concrete table inheritance works perfectly here as it provides separate and isolated tables, and because the `position` field must be managed for records that share the same type.

Build the model and see what happens. Doctrine and symfony have generated three SQL tables and six model classes in the `lib/model/doctrine` directory:

- `sfReferential`: manages the `sf_referential` records,
- `sfReferentialTable`: manages the `sf_referential` table,

- `sfReferentialContractType`: manages the `sf_referential_contract_type` records.
- `sfReferentialContractTypeTable`: manages the `sf_referential_contract_type` table.
- `sfReferentialProductType`: manages the `sf_referential_product_type` records.
- `sfReferentialProductTypeTable`: manages the `sf_referential_product_type` table.

Exploring the generated inheritance shows that both base classes of the `sfReferentialContractType` and `sfReferentialProductType` model classes inherit from the `sfReferential` class. So, all protected and public methods (including properties) placed in the `sfReferential` class will be shared amongst the two subclasses and can be overridden if necessary.

That's exactly the expected goal. The `sfReferential` class can now contain methods to manage all referential data, for example:

*Listing 9-7*

```
// lib/model/doctrine/sfReferential.class.php
class sfReferential extends BasesfReferential
{
  public function promote()
  {
    // move up the record in the list
  }

  public function demote()
  {
    // move down the record in the list
  }

  public function moveToFirstPosition()
  {
    // move the record to the first position
  }

  public function moveToLastPosition()
  {
    // move the record to the last position
  }

  public function moveToPosition($position)
  {
    // move the record to a given position
  }

  public function makeDefault($forceSave = true, $conn = null)
  {
    $this->setIsDefault(true);

    if ($forceSave)
    {
      $this->save($conn);
    }
  }
}
```

Thanks to the Doctrine concrete table inheritance, all the code is shared in the same place. Code becomes easier to debug, maintain, improve and unit test.

That's the first real advantage when dealing with table inheritance. Additionally, thanks to this approach, model objects can be used to centralize actions code as shown below. The `sfBaseReferentialActions` is a special actions class inherited by each actions class that manages a referential model.

*Listing 9-8*

```
// lib/actions/sfBaseReferentialActions.class.php
class sfBaseReferentialActions extends sfActions
{
  /**
   * Ajax action that saves the new position as a result of the user
   * using a drag and drop in the list view.
   *
   * This action is linked thanks to an ~sfDoctrineRoute~ that
   * eases single referential object retrieval.
   *
   * @param sfWebRequest $request
   */
  public function executeMoveToPosition(sfWebRequest $request)
  {
    $this->forward404Unless($request->isXmlHttpRequest());

    $referential = $this->getRoute()->getObject();

    $referential->moveToPosition($request->getParameter('position', 1));

    return sfView::NONE;
  }
}
```

What would happen if the schema did not use table inheritance? The code would need to be duplicated in each referential subclass. This approach wouldn't be DRY, (Don't Repeat Yourself) especially for an application with a dozen referential tables.

## Table Inheritance at the Forms Layer

Let's continue the guided tour of Doctrine table inheritance's advantages. The previous section demonstrated how this feature can be really useful to share methods and properties between several inherited models. Let's have a look now at how it behaves when dealing with symfony generated forms.

### The Study Case's Model

The YAML schema below describes a model to manage digital documents. The aim is to store generic information in the `File` table and specific data in sub-tables like `Video` and `PDF`.

*Listing 9-9*

```
File:
  columns:
    filename:
      type:           string(50)
      notnull:        true
    mime_type:
      type:           string(50)
      notnull:        true
    description:
      type:           clob
```

```
      notnull:         true
    size:
      type:            integer(8)
      notnull:         true
      default:         0

Video:
  inheritance:
    type:              concrete
    extends:           File
  columns:
    format:
      type:            string(30)
      notnull:         true
    duration:
      type:            integer(8)
      notnull:         true
      default:         0
    encoding:
      type:            string(50)

PDF:
  tableName:           pdf
  inheritance:
    type:              concrete
    extends:           File
  columns:
    pages:
      type:            integer(8)
      notnull:         true
      default:         0
    paper_size:
      type:            string(30)
    orientation:
      type:            enum
      default:         portrait
      values:          [portrait, landscape]
    is_encrypted:
      type:            boolean
      default:         false
      notnull:         true
```

Both the `PDF` and `Video` tables share the same `File` table, which contains global information about digital files. The `Video` model encapsulates data related to video objects such as `format` (4/3, 16/9...) or `duration`, whereas the `PDF` model contains the number of `pages` or the document's `orientation`. Let's build this model and generate the corresponding forms.

```
$ php symfony doctrine:build --all
```

*Listing 9-10*

The following section describes how to take advantage of the table inheritance in form classes thanks to the new `setupInheritance()` method.

## Discover the setupInheritance() method

As expected, Doctrine has generated six form classes in the `lib/form/doctrine` and `lib/form/doctrine/base` directories:

- `BaseFileForm`

- `BaseVideoForm`
- `BasePDFForm`
- `FileForm`
- `VideoForm`
- `PDFForm`

Let's open the three `Base` form classes to discover something new in the `setup()` method. A new `setupInheritance()` method has been added for symfony 1.3. This method is empty by default.

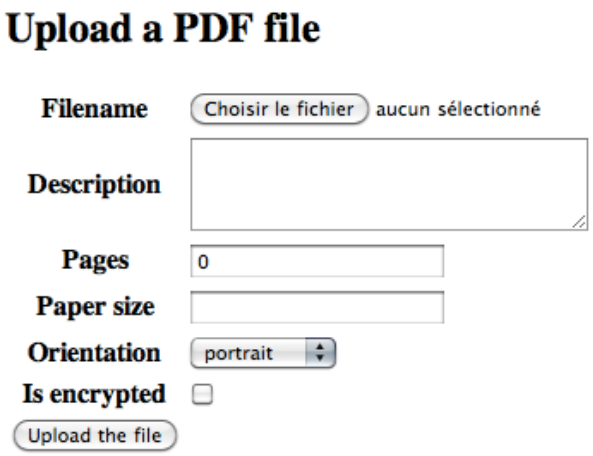
The most important thing to notice is that the form inheritance is preserved as `BaseVideoForm` and `BasePDFForm` both extend the `FileForm` and `BaseFileForm` classes. Consequently, each inherits from the `File` class and can share the same base methods.

The following listing overrides the `setupInheritance()` method and configures the `FileForm` class so that it can be used in either subform more effectively.

*Listing 9-11*

```
// lib/form/doctrine/FileForm.class.php
class FileForm extends BaseFileForm
{
  protected function setupInheritance()
  {
    parent::setupInheritance();

    $this->useFields(array('filename', 'description'));

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

The `setupInheritance()` method, which is called by both the `VideoForm` and `PDFForm` subclasses, removes all fields except `filename` and `description`. The `filename` field's widget has been turned into a file widget and its corresponding validator has been changed to an `sfValidatorFile` validator. This way, the user will be able to upload a file and save it to the server.



### Set the Current File's Mime Type and Size

All the forms are now ready and customized. There is one more thing to configure, however, before being able to use them. As the `mime_type` and `size` fields have been removed from the `FileForm` object, they must be set programmatically. The best place to do this is in a new `generateFilenameFilename()` method in the `File` class.

*Listing 9-12*

```
// lib/model/doctrine/File.class.php
class File extends BaseFile
{
  /**
   * Generates a filename for the current file object.
   *
   * @param sfValidatedFile $file
   * @return string
   */
  public function generateFilenameFilename(sfValidatedFile $file)
  {
    $this->setMimeType($file->getType());
    $this->setSize($file->getSize());

    return $file->generateFilename();
  }
}
```

This new method aims to generate a custom filename for the file to store on the file system. Although the `generateFilenameFilename()` method returns a default auto-generated filename, it also sets the `mime_type` and `size` properties on the fly thanks to the `sfValidatedFile` object passed as its first argument.

As symfony 1.3 entirely supports Doctrine table inheritance, forms are now able to save an object and its inherited values. The native inheritance support allows for powerful and functional forms with very few chunks of customized code.

The above example could be widely and easily improved thanks to class inheritance. For example, both the `VideoForm` and `PDFForm` classes could override the `filename` validator to a more specific custom validator such as `sfValidatorVideo` or `sfValidatorPDF`.

## Table Inheritance at the Filters Layer

Because filters are also forms, they too inherit the methods and properties of parent form filters. Consequently, the `VideoFormFilter` and `PDFFormFilter` objects extend the `FileFormFilter` class and can be customized by using the `setupInheritance()` method.

In the same way, both `VideoFormFilter` and `PDFFormFilter` can share the same custom methods in the `FileFormFilter` class.

## Table Inheritance at the Admin Generator Layer

It's now time to discover how to take advantage of Doctrine table inheritance as well as one of the Admin Generator's new features: the **actions base class** definition. The Admin Generator is one of the most improved features of symfony since the 1.0 version.

In November 2008, symfony introduced the new Admin Generator system bundled with version 1.2. This tool comes with a lot of functionality out of the box such as basic CRUD operations, list filtering and paging, batch deleting and so on... The Admin Generator is a powerful tool, which eases and accelerates backend generation and customization for any developer.

## Practical Example Introduction

The aim of the last part of this chapter is to illustrate how to take advantage of the Doctrine table inheritance coupled with the Admin Generator. To achieve this, a simple backend area will be constructed to manage two tables, which both contain data that can be sorted / prioritized.

As symfony's mantra is to not reinvent the wheel every time, the Doctrine model will use the csDoctrineActAsSortablePlugin[65] to provide all the needed API to sort objects between each other. The `csDoctrineActAsSortablePlugin` plugin is developed and maintained by CentreSource, one of the most active companies in the symfony ecosystem.

The data model is quite simple. There are three model classes, `sfItem`, `sfTodoItem` and `sfShoppingItem`, which help to manage a todo list and a shopping list. Each item in both lists is sortable to allow items to be prioritize within the list.

*Listing 9-13*

```
sfItem:
  actAs:              [Timestampable]
  columns:
    name:
      type:           string(50)
      notnull:        true

sfTodoItem:
  actAs:              [Sortable]
  inheritance:
    type:             concrete
    extends:          sfItem
  columns:
    priority:
      type:           string(20)
      notnull:        true
      default:        minor
    assigned_to:
      type:           string(30)
      notnull:        true
      default:        me

sfShoppingItem:
  actAs:              [Sortable]
  inheritance:
    type:             concrete
    extends:          sfItem
  columns:
    quantity:
      type:           integer(3)
      notnull:        true
      default:        1
```

The above schema describes the data model split in three model classes. The two children classes (`sfTodoItem`, `sfShoppingItem`) both use the `Sortable` and `Timestampable` behaviors. The `Sortable` behavior is provided by the `csDoctrineActAsSortablePlugin` plugin and adds a integer `position` column to each table. Both classes extend the `sfItem` base class. This class contains an `id` and `name` column.

65. http://www.symfony-project.org/plugins/csDoctrineActAsSortablePlugin

Let's add some data fixtures so that we have some test data to play with inside the backend. The data fixtures are, as usual, located in the `data/fixtures.yml` file of the symfony project.

Listing 9-14

```
sfTodoItem:
  sfTodoItem_1:
    name:           "Write a new symfony book"
    priority:       "medium"
    assigned_to:    "Fabien Potencier"
  sfTodoItem_2:
    name:           "Release Doctrine 2.0"
    priority:       "minor"
    assigned_to:    "Jonathan Wage"
  sfTodoItem_3:
    name:           "Release symfony 1.4"
    priority:       "major"
    assigned_to:    "Kris Wallsmith"
  sfTodoItem_4:
    name:           "Document Lime 2 Core API"
    priority:       "medium"
    assigned_to:    "Bernard Schussek"

sfShoppingItem:
  sfShoppingItem_1:
    name:           "Apple MacBook Pro 15.4 inches"
    quantity:       3
  sfShoppingItem_2:
    name:           "External Hard Drive 320 GB"
    quantity:       5
  sfShoppingItem_3:
    name:           "USB Keyboards"
    quantity:       2
  sfShoppingItem_4:
    name:           "Laser Printer"
    quantity:       1
```

Once the `csDoctrineActAsSortablePlugin` plugin is installed and the data model is ready, the new plugin needs to be activated in the `ProjectConfiguration` class located in `config/ProjectConfiguration.class.php`:

Listing 9-15

```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->enablePlugins(array(
      'sfDoctrinePlugin',
      'csDoctrineActAsSortablePlugin'
    ));
  }
}
```

Next, the database, model, forms and filters can be generated and the fixtures loaded into the database to feed the newly created tables. This can be accomplished at once thanks to the `doctrine:build` task:

Listing 9-16

```
$ php symfony doctrine:build --all --no-confirmation
```

The symfony cache must be cleared to complete the process and the plugin's assets have to be linked inside the `web` directory:

Listing 9-17
```
$ php symfony cache:clear
$ php symfony plugin:publish-assets
```

The following section explains how to build the backend modules with the Admin Generator tools and how to benefit from the new actions base class feature.

## Setup The Backend

This section describes the steps required to setup a new backend application containing two generated modules that manage both the shopping and todo lists. Consequently, the first thing to do is to generate a `backend` application to house the coming modules:

Listing 9-18
```
$ php symfony generate:app backend
```

Even though the Admin Generator is a great tool, prior to symfony 1.3, the developer was forced to duplicate common code between generated modules. Now, however, the `doctrine:generate-admin` task introduces a new `--actions-base-class` option that allows the developer to define the module's base actions class.

As the two modules are quiet similar, they will certainly need to share some generic actions code. This code can be located in a super actions class located in the `lib/actions` directory as shown in the code below:

Listing 9-19
```
// lib/actions/sfSortableModuleActions.class.php
class sfSortableModuleActions extends sfActions
{

}
```

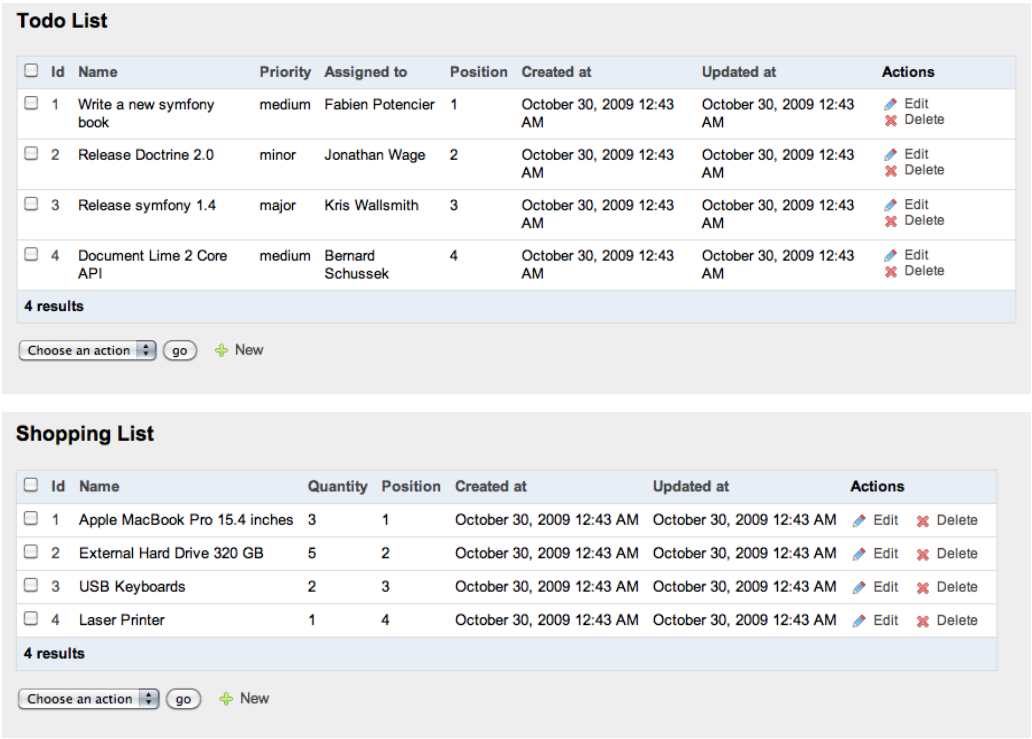
Once the new `sfSortableModuleActions` class is created and the cache has been cleared, the two modules can be generated in the backend application:

Listing 9-20
```
$ php symfony doctrine:generate-admin --module=shopping
--actions-base-class=sfSortableModuleActions backend sfShoppingItem
```

Listing 9-21
```
$ php symfony doctrine:generate-admin --module=todo
--actions-base-class=sfSortableModuleActions backend sfTodoItem
```

The Admin Generator generates modules in two separate directories. The first directory is, of course, `apps/backend/modules`. The majority of the generated module files, however, are located in the `cache/backend/dev/modules` directory. Files located in this location are regenerated each time the cache is cleared or when the module's configuration changes.

Browsing the cached files is a great way to understand how symfony and the Admin Generator work together under the hood. Consequently, the new `sfSortableModuleActions` subclasses can be found in `cache/backend/dev/modules/autoShopping/actions/actions.class.php` and `cache/backend/dev/modules/autoTodo/actions/actions.class.php`. By default, symfony would generate these classes to inherit directly from `sfActions`.




The two backend modules are ready to be used and customized. It's not the goal of this chapter, however, to explore the configuration of auto generated modules. Significant documentation exists on this topic, including in the symfony Reference Book[66].

## Changing an Item's Position

The previous section described how to setup two fully functional backend modules, which both inherit from the same actions class. The next goal is to create a shared action, which allows the developer to sort objects from a list between each other. This is quite easy as the installed plugin provides a full API to handle the resorting of the objects.

The first step is to create two new routes capable of moving a record up or down in the list. As the Admin Generator uses the `sfDoctrineRouteCollection` route, new routes can be easily declared and attached to the collection via the `config/generator.yml` of both modules:

*Listing 9-22*

```
# apps/backend/modules/shopping/config/generator.yml
generator:
  class: sfDoctrineGenerator
  param:
    model_class:           sfShoppingItem
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          sf_shopping_item
    with_doctrine_route:   true
    actions_base_class:    sfSortableModuleActions

    config:
      actions: ~
      fields:  ~
```

66. http://www.symfony-project.org/reference/1_3/en/06-Admin-Generator

```
      list:
        max_per_page:      100
        sort:              [position, asc]
        display:           [position, name, quantity]
        object_actions:
          moveUp:          { label: "move up", action: "moveUp" }
          moveDown:        { label: "move down", action: "moveDown" }
          _edit:      ~
          _delete:    ~
      filter:  ~
      form:    ~
      edit:    ~
      new:     ~
```

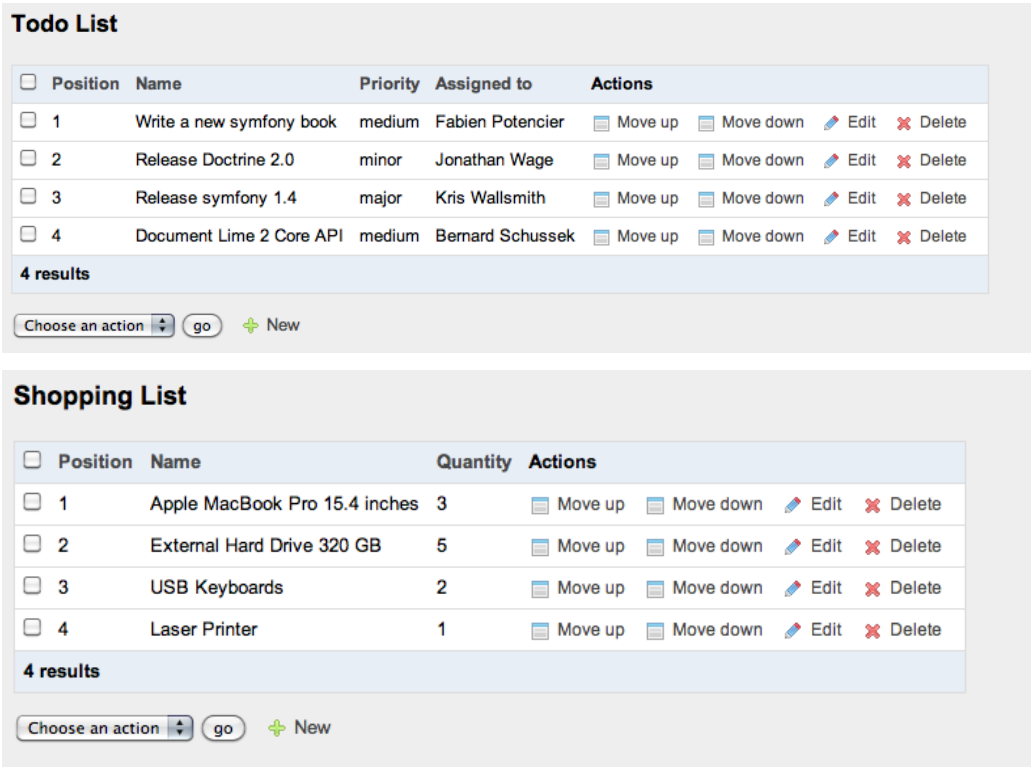
Changes need to be repeated for the `todo` module:

*Listing 9-23*

```
# apps/backend/modules/todo/config/generator.yml
generator:
  class: sfDoctrineGenerator
  param:
    model_class:           sfTodoItem
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          sf_todo_item
    with_doctrine_route:   true
    actions_base_class:    sfSortableModuleActions

    config:
      actions: ~
      fields:  ~
      list:
        max_per_page:      100
        sort:              [position, asc]
        display:           [position, name, priority, assigned_to]
        object_actions:
          moveUp:          { label: "move up", action: "moveUp" }
          moveDown:        { label: "move down", action: "moveDown" }
          _edit:      ~
          _delete:    ~
      filter:  ~
      form:    ~
      edit:    ~
      new:     ~
```

The two YAML files describe the configuration for both `shopping` and `todo` modules. Each of these has been customized to fit the end user's needs. First, the list view is ordered on the `position` column with an `ascending` ordering. Next, the number of max items per page has been increased to 100 to avoid pagination.

Finally, the number of displayed columns has been reduced to the `position`, `name`, `priority`, `assigned_to` and `quantity` columns. Additionally, each module has two new actions: `moveUp` and `moveDown`. The final rendering should look like the following screenshots:




These two new actions have been declared but for now don't do anything. Each must be created in the shared actions class, `sfSortableModuleActions` as described below. The `csDoctrineActAsSortablePlugin` plugin provides two extra useful methods on each model class: `promote()` and `demote()`. Each is used to build the `moveUp` and `moveDown` actions.

*Listing 9-24*

```
// lib/actions/sfSortableModuleActions.class.php
class sfSortableModuleActions extends sfActions
{
  /**
   * Moves an item up in the list.
   *
   * @param sfWebRequest $request
   */
  public function executeMoveUp(sfWebRequest $request)
  {
    $this->item = $this->getRoute()->getObject();

    $this->item->promote();

    $this->redirect($this->getModuleName());
  }

  /**
   * Moves an item down in the list.
   *
   * @param sfWebRequest $request
   */
  public function executeMoveDown(sfWebRequest $request)
  {
    $this->item = $this->getRoute()->getObject();

    $this->item->demote();
```

```
    $this->redirect($this->getModuleName());
  }
}
```

Thanks to these two shared actions, both the todo list and the shopping list are sortable. Moreover, they are easy to maintain and test with functional tests. Feel free to improve the look and feel of both modules by overriding the object's actions template to remove the first `move up` link and the last `move down` link.

## Special Gift: Improve the User's Experience

Before finishing, let's polish the two lists to improve the user's experience. Everybody agrees that moving a record up (or down) by clicking a link is not really intuitive for the end user. A better approach is definitively to include JavaScript ajax behaviors. In this case, all HTML table rows will be draggable and droppable thanks to the `Table Drag and Drop` jQuery plugin. An ajax call will be performed whenever the user moves a row in the HTML table.

First grab and install the jQuery framework under the `web/js` directory and then repeat the operation for the `Table Drag and Drop` plugin, whose source code is hosted on a Google Code[67] repository.

To work, the list view of each module must include a little JavaScript snippet and both tables need an `id` attribute. As all admin generator templates and partials can be overridden, the `_list.php` file, located in the cache by default, should be copied to both modules.

But wait, copying the `_list.php` file under the `templates/` directory of each module is not really DRY. Just copy the `cache/backend/dev/modules/autoShopping/templates/_list.php` file to the `apps/backend/templates/` directory and rename it `_table.php`. Replace its current content with the following code:

*Listing 9-25*

```
<div class="sf_admin_list">
  <?php if (!$pager->getNbResults()): ?>
    <p><?php echo __('No result', array(), 'sf_admin') ?></p>
  <?php else: ?>
    <table cellspacing="0" id="sf_item_table">
      <thead>
        <tr>
          <th id="sf_admin_list_batch_actions"><input
id="sf_admin_list_batch_checkbox" type="checkbox" onclick="checkAll();"
/></th>
          <?php include_partial(
            $sf_request->getParameter('module').'/list_th_tabular',
            array('sort' => $sort)
          ) ?>
          <th id="sf_admin_list_th_actions">
            <?php echo __('Actions', array(), 'sf_admin') ?>
          </th>
        </tr>
      </thead>
      <tfoot>
        <tr>
          <th colspan="<?php echo $colspan ?>">
            <?php if ($pager->haveToPaginate()): ?>
              <?php include_partial(
                $sf_request->getParameter('module').'/pagination',
                array('pager' => $pager)
```

67. http://code.google.com/p/tablednd/

```
              ) ?>
            <?php endif; ?>
            <?php echo format_number_choice(
              '[0] no result|[1] 1 result|(1,+Inf] %1% results',
              array('%1%' => $pager->getNbResults()),
              $pager->getNbResults(), 'sf_admin'
            ) ?>
            <?php if ($pager->haveToPaginate()): ?>
              <?php echo __('(page %%page%%/%%nb_pages%%)', array(
                '%%page%%' => $pager->getPage(),
                '%%nb_pages%%' => $pager->getLastPage()),
                'sf_admin'
              ) ?>
            <?php endif; ?>
          </th>
        </tr>
      </tfoot>
      <tbody>
      <?php foreach ($pager->getResults() as $i => $item): ?>
        <?php $odd = fmod(++$i, 2) ? 'odd' : 'even' ?>
        <tr class="sf_admin_row <?php echo $odd ?>">
          <?php include_partial(
            $sf_request->getParameter('module').'/list_td_batch_actions',
            array(
              'sf_'. $sf_request->getParameter('module') .'_item' => $item,
              'helper' => $helper
          )) ?>
          <?php include_partial(
            $sf_request->getParameter('module').'/list_td_tabular',
            array(
              'sf_'. $sf_request->getParameter('module') .'_item' => $item
          )) ?>
            <?php include_partial(
              $sf_request->getParameter('module').'/list_td_actions',
              array(
                'sf_'. $sf_request->getParameter('module') .'_item' =>
$item,
                'helper' => $helper
            )) ?>
        </tr>
      <?php endforeach; ?>
      </tbody>
    </table>
  <?php endif; ?>
  </div>
  <script type="text/javascript">
    /* <![CDATA[ */
    function checkAll() {
      var boxes = document.getElementsByTagName('input');
      for (var index = 0; index < boxes.length; index++) {
        box = boxes[index];
        if (
          box.type == 'checkbox'
          &&
          box.className == 'sf_admin_batch_checkbox'
        )
        box.checked =
document.getElementById('sf_admin_list_batch_checkbox').checked
```

```
      }
      return true;
    }
    /* ]]> */
  </script>
```

Finally, create a `_list.php` file inside each module's `templates` directory, and place the following code in each:

Listing 9-26
```
// apps/backend/modules/shopping/templates/_list.php
<?php include_partial('global/table', array(
  'pager' => $pager,
  'helper' => $helper,
  'sort' => $sort,
  'colspan' => 5
)) ?>
```

Listing 9-27
```
// apps/backend/modules/shopping/templates/_list.php
<?php include_partial('global/table', array(
  'pager' => $pager,
  'helper' => $helper,
  'sort' => $sort,
  'colspan' => 8
)) ?>
```

To change the position of a row, both modules need to implement a new action that processes the coming ajax request. As seen before, the new shared `executeMove()` action will be placed in the `sfSortableModuleActions` actions class:

Listing 9-28
```
// lib/actions/sfSortableModuleActions.class.php
class sfSortableModuleActions extends sfActions
{
  /**
   * Performs the Ajax request, moves an item to a new position.
   *
   * @param sfWebRequest $request
   */
  public function executeMove(sfWebRequest $request)
  {
    $this->forward404Unless($request->isXmlHttpRequest());
    $this->forward404Unless($item =
Doctrine_Core::getTable($this->configuration->getModel())->find($request->getParameter(

    $item->moveToPosition((int) $request->getParameter('rank', 1));

    return sfView::NONE;
  }
}
```

The `executeMove()` action requires a `getModel()` method on the configuration object. Implement this new method in both the `todoGeneratorConfiguration` and `shoppingGeneratorConfiguration` classes as shown below:

Listing 9-29
```
// apps/backend/modules/shopping/lib/
shoppingGeneratorConfiguration.class.php
class shoppingGeneratorConfiguration extends
BaseShoppingGeneratorConfiguration
{
```

```
  public function getModel()
  {
    return 'sfShoppingItem';
  }
}
```

Listing 9-30

```
// apps/backend/modules/todo/lib/todoGeneratorConfiguration.class.php
class todoGeneratorConfiguration extends BaseTodoGeneratorConfiguration
{
  public function getModel()
  {
    return 'sfTodoItem';
  }
}
```

There is one last operation to do. For now, the tables rows are not draggable and no ajax call is performed when a moved row is released. To achieve this, both modules need a specific route to access their corresponding `move` action. Consequently, the `apps/backend/config/routing.yml` file needs the following two new routes as shown below:

Listing 9-31

```
<?php foreach (array('shopping', 'todo') as $module) : ?>

<?php echo $module ?>_move:
  class: sfRequestRoute
  url: /<?php echo $module ?>/move
  param:
    module: "<?php echo $module ?>"
    action: move
  requirements:
    sf_method: [get]

<?php endforeach ?>
```

To avoid code duplication, the two routes are generated inside a `foreach` statement and are based on the module name to easily retrieve it in the view. Finally, the `apps/backend/templates/_table.php` must implement the JavaScript snippet in order to initialize the drag and drop behavior and the corresponding ajax request:

Listing 9-32

```
<script type="text/javascript" charset="utf-8">
  $().ready(function() {
    $("#sf_item_table").tableDnD({
      onDrop: function(table, row) {
        var rows = table.tBodies[0].rows;

        // Get the moved item's id
        var movedId = $(row).find('td input:checkbox').val();

        // Calculate the new row's position
        var pos = 1;
        for (var i = 0; i<rows.length; i++) {
          var cells = rows[i].childNodes;
          // Perform the ajax request for the new position
          if (movedId == $(cells[1]).find('input:checkbox').val()) {
            $.ajax({
              url:"<?php echo url_for('@'.
$sf_request->getParameter('module').'_move') ?>?id="+ movedId +"&rank="+
pos,
              type:"GET"
```

```
            });
            break;
          }
          pos++;
        }
      },
    });
  });
</script>
```

The HTML table is now fully functional. Rows are draggable and droppable, and the new position of a row is automatically saved thanks to an ajax call. With just a few code chunks, the backend's usability has been greatly improved to offer the end user a better experience. The Admin Generator is flexible enough to be extended and customized and works perfectly with Doctrine's table inheritance.

Feel free to improve the two modules by removing the two obsolete `moveUp` and `moveDown` actions and adding any other customizations that fit your needs.

# Final Thoughts

This chapter described how Doctrine table inheritance is a powerful feature, which helps the developer code faster and improve code organization. This Doctrine functionality is fully integrated at several levels in symfony. Developers are encouraged to take advantage of it to increase efficiency and promote code organization.

Chapter 10

# Symfony Internals

*by Geoffrey Bachelet*

Have you ever wondered what happens to a HTTP request when it reaches a symfony application? If yes, then you are in the right place. This chapter will explain in depth how symfony processes each request in order to create and return the response. Of course, just describing the process would lack a bit of fun, so we'll also have a look at some interesting things you can do and where you can interact with this process.

## The Bootstrap

It all begins in your application's controller. Say you have a `frontend` controller with a `dev` environment (a very classic start for any symfony project). In this case, you'll end up with a front controller located at `web/frontend_dev.php`[68]. What exactly happens in this file? In just a few lines of code, symfony retrieves the application configuration and creates an instance of `sfContext`, which is responsible for dispatching the request. The application configuration is necessary when creating the `sfContext` object, which is the application-dependent engine behind symfony.

Symfony already gives you quite a bit of control on what happens here allowing you to pass a custom root directory for your application as the fourth argument of `ProjectConfiguration::getApplicationConfiguration()` as well as a custom context class as the third (and last) argument of `sfContext::createInstance()`[69] (but remember it has to extend `sfContext`).

Retrieving the application's configuration is a very important step. First, `sfProjectConfiguration` is responsible for guessing the application's configuration class, usually `${application}Configuration`, located in `apps/${application}/config/${application}Configuration.class.php`.

`sfApplicationConfiguration` actually extends `ProjectConfiguration`, meaning that any method in `ProjectConfiguration` can be shared between all applications. This also means that `sfApplicationConfiguration` shares its constructor with both `ProjectConfiguration` and `sfProjectConfiguration`. This is fortunate since much of the project is configured inside the `sfProjectConfiguration` constructor. First, several useful values are computed and stored, such as the project's root directory and the symfony library directory. `sfProjectConfiguration` also creates a new event dispatcher of type

68. `http://trac.symfony-project.org/browser/branches/1.3/lib/task/generator/skeleton/app/web/index.php`
69. `http://www.symfony-project.org/api/1_3/sfContext#method_createinstance`

`sfEventDispatcher`, unless one was passed as the fifth argument of `ProjectConfiguration::getApplicationConfiguration()` in the front controller.

Just after that, you are given a chance to interact with the configuration process by overriding the `setup()` method of `ProjectConfiguration`. This is usually the best place to enable / disable plugins (using `sfProjectConfiguration::setPlugins()`[70], `sfProjectConfiguration::enablePlugins()`[71], `sfProjectConfiguration::disablePlugins()`[72] or `sfProjectConfiguration::enableAllPluginsExcept()`[73]).

Next the plugins are loaded by `sfProjectConfiguration::loadPlugins()`[74] and the developer has a chance to interact with this process through the `sfProjectConfiguration::setupPlugins()`[75] that can be overriden.

Plugin initialization is quite straight forward. For each plugin, symfony looks for a `${plugin}Configuration` (e.g. `sfGuardPluginConfiguration`) class and instantiates it if found. Otherwise, `sfPluginConfigurationGeneric` is used. You can hook into a plugin's configuration through two methods:

- `${plugin}Configuration::configure()`, before autoloading is done
- `${plugin}Configuration::initialize()`, after autoloading

Next, `sfApplicationConfiguration` executes its `configure()` method, which can be used to customize each application's configuration before the bulk of the internal configuration initialization process begins in `sfApplicationConfiguration::initConfiguration()`[76].

This part of symfony's configuration process is responsible for many things and there are several entry points if you want to hook into this process. For example, you can interact with the autoloader's configuration by connecting to the `autoload.filter_config` event. Next, several very important configuration files are loaded, including `settings.yml` and `app.yml`. Finally, a last bit of plugin configuration is available through each plugin's `config/config.php` file or configuration class's `initialize()` method.

If `sf_check_lock` is activated, symfony will now check for a lock file (the one created by the `project:disable` task, for example). If the lock is found, the following files are checked and the first available is included, followed immediately by termination of the script:

1. `apps/${application}/config/unavailable.php`,
2. `config/unavailable.php`,
3. `web/errors/unavailable.php`,
4. `lib/vendor/symfony/lib/exception/data/unavailable.php`,

Finally, the developer has one last chance to customize the application's initialization through the `sfApplicationConfiguration::initialize()` method.

---

70. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_setplugins
71. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_enableplugins
72. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_disableplugins
73. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_enableallpluginsexcept
74. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_loadplugins
75. http://www.symfony-project.org/api/1_3/sfProjectConfiguration#method_setupplugins
76. http://www.symfony-project.org/api/1_3/sfApplicationConfiguration#method_initconfiguration

### Bootstrap and configuration summary

- Retrieval of the application's configuration
  - `ProjectConfiguration::setup()` (define your plugins here)
  - Plugins are loaded
  - `${plugin}Configuration::configure()`
  - `${plugin}Configuration::initialize()`
  - `ProjectConfiguration::setupPlugins()` (setup your plugins here)
  - `${application}Configuration::configure()`
  - `autoload.filter_config` is notified
  - Loading of `settings.yml` and `app.yml`
  - `${application}Configuration::initialize()`
- Creation of an `sfContext` instance

## `sfContext` and Factories

Before diving into the dispatch process, let's talk about a vital part of the symfony workflow: the factories.

In symfony, factories are a set of components or classes that your application relies on. Examples of factories are `logger`, `i18n`, etc. Each factory is configured via `factories.yml`, which is compiled by a config handler (more on config handlers later) and converted into PHP code that actually instantiates the factory objects (you can view this code in your cache in the `cache/frontend/dev/config/config_factories.yml.php` file).

Factory loading happens upon `sfContext` initialization. See `sfContext::initialize()`[77] and `sfContext::loadFactories()`[78] for more information.

At this point, you can already customize a large part of symfony's behavior just by editing the `factories.yml` configuration. You can even replace symfony's built-in factory classes with your own!

If you're interested in knowing more about factories, The symfony reference book[79] as well as the `factories.yml`[80] file itself are invaluable resources.

If you looked at the generated `config_factories.yml.php`, you may have noticed that factories are instantiated in a certain order. That order is important since some factories are dependent on others (for example, the `routing` component obviously needs the `request` to retrieve the information it needs).

Let's talk in greater details about the `request`. By default, the `sfWebRequest` class represents the `request`. Upon instantiation, `sfWebRequest::initialize()`[81] is called, which gathers relevant information such as the GET / POST parameters as well as the HTTP method. You're then given an opportunity to add your own request processing through the `request.filter_parameters` event.

77. `http://www.symfony-project.org/api/1_3/sfContext#method_initialize`
78. `http://www.symfony-project.org/api/1_3/sfContext#method_loadfactories`
79. `http://www.symfony-project.org/reference/1_3/en/05-Factories`
80. `http://trac.symfony-project.org/browser/branches/1.3/lib/config/config/factories.yml`
81. `http://www.symfony-project.org/api/1_3/sfWebRequest#method_initialize`

### Using the `request.filter_parameter` event

Let's say you're operating a website exposing a public API to your users. The API is available through HTTP, and each user wanting to use it must provide a valid API key through a request header (for example `X_API_KEY`) to be validated by your application. This can be easily achieved using the `request.filter_parameter` event:

*Listing 10-1*
```
class apiConfiguration extends sfApplicationConfiguration
{
  public function configure()
  {
    // ...

    $this->dispatcher->connect('request.filter_parameters', array(
      $this, 'requestFilterParameters'
    ));
  }

  public function requestFilterParameters(sfEvent $event, $parameters)
  {
    $request = $event->getSubject();

    $api_key = $request->getHttpHeader('X_API_KEY');

    if (null === $api_key || false === $api_user =
Doctrine_Core::getTable('ApiUser')->findOneByToken($api_key))
    {
      throw new RuntimeException(sprintf('Invalid api key "%s"',
$api_key));
    }

    $request->setParameter('api_user', $api_user);

    return $parameters;
  }
}
```

You will then be able to access your API user from the request:

*Listing 10-2*
```
public function executeFoobar(sfWebRequest $request)
{
  $api_user = $request->getParameter('api_user');
}
```

This technique can be used, for example, to validate webservice calls.

The `request.filter_parameters` event comes with a lot of information about the request, see the `sfWebRequest::getRequestContext()`[82] method for more information.

The next very important factory is the routing. Routing's initialization is fairly straightforward and consists mostly of gathering and setting specific options. You can, however, hook up to this process through the `routing.load_configuration` event.

82. `http://www.symfony-project.org/api/1_3/sfWebRequest#method_getrequestcontext`

The `routing.load_configuration` event gives you access to the current routing object's instance (by default, `sfPatternRouting`[83]). You can then manipulate registered routes through a variety of methods.

### `routing.load_configuration` event usage example

For example, you can easily add a route:

*Listing 10-3*

```
public function setup()
{
  // ...

  $this->dispatcher->connect('routing.load_configuration', array(
    $this, 'listenToRoutingLoadConfiguration'
  ));
}

public function listenToRoutingLoadConfiguration(sfEvent $event)
{
  $routing = $event->getSubject();

  if (!$routing->hasRouteName('my_route'))
  {
    $routing->prependRoute('my_route', new sfRoute(
      '/my_route', array('module' => 'default', 'action' => 'foo')
    ));
  }
}
```

URL parsing occurs right after initialization, via the `sfPatternRouting::parse()`[84] method. There are quite a few methods involved, but it suffices to say that by the time we reach the end of the `parse()` method, the correct route has been found, instantiated and bound to relevant parameters.

For more information about routing, please see the `Advanced Routing` chapter of this book.

Once all factories have been loaded and properly setup, the `context.load_factories` event is triggered. This event is important since it's the earliest event in the framework where the developer has access to all of symfony's core factory objects (request, response, user, logging, database, etc.).

This is also the time to connect to another very useful event: `template.filter_parameters`. This event occurs whenever a file is rendered by `sfPHPView`[85] and allows the developer to control the parameters actually passed to the template. `sfContext` takes advantage of this event to add some useful parameters to each template (namely, `$sf_context`, `$sf_request`, `$sf_params`, `$sf_response` and `$sf_user`).

83. `http://trac.symfony-project.org/browser/branches/1.3/lib/routing/sfPatternRouting.class.php`
84. `http://www.symfony-project.org/api/1_3/sfPatternRouting#method_parse`
85. `http://trac.symfony-project.org/browser/branches/1.3/lib/view/sfPHPView.class.php`

You can connect to the `template.filter_parameters` event in order to add additional custom global parameters to all templates.

### Taking advantage of the `template.filter_parameters` event

Say you decide that every single template you use should have access to a particular object, say a helper object. You would then add the following code to `ProjectConfiguration`:

*Listing 10-4*

```
public function setup()
{
  // ...

  $this->dispatcher->connect('template.filter_parameters', array(
    $this, 'templateFilterParameters'
  ));
}

public function templateFilterParameters(sfEvent $event, $parameters)
{
  $parameters['my_helper_object'] = new MyHelperObject();

  return $parameters;
}
```

Now every template has access to an instance of `MyHelperObject` through `$my_helper_object`.

### `sfContext` summary

1. Initialization of `sfContext`
2. Factory loading
3. Events notified:
    1. request.filter_parameters[86]
    2. routing.load_configuration[87]
    3. context.load_factories[88]
4. Global templates parameters added

## A Word on Config Handlers

Config handlers are at the heart of symfony's configuration system. A config handler is tasked with *understanding* the meaning behind a configuration file. Each config handler is simply a class that is used to translate a set of yaml configuration files into a block of PHP code that can be executed as needed. Each configuration file is assigned to one specific config handler in the `config_handlers.yml` file[89].

To be clear, the job of a config handler is *not* to actually parse the yaml files (this is handled by `sfYaml`). Instead each config handler creates a set of PHP directions based on the YAML

86. `http://www.symfony-project.org/reference/1_3/en/15-Events#chapter_15_sub_request_filter_parameters`
87. `http://www.symfony-project.org/reference/1_3/en/15-Events#chapter_15_sub_routing_load_configuration`
88. `http://www.symfony-project.org/reference/1_3/en/15-Events#chapter_15_sub_context_load_factories`
89. `http://trac.symfony-project.org/browser/branches/1.3/lib/config/config/config_handlers.yml`

information and saves those directions to a PHP file, which can be efficiently included later. The *compiled* version of each YAML configuration file can be found in the cache directory.

The most commonly used config handler is most certainly `sfDefineEnvironmentConfigHandler`[90], which allows for environment-specific configuration settings. This config handler takes care to fetch only the configuration settings of the current environment.

Still not convinced? Let's explore `sfFactoryConfigHandler`[91]. This config handler is used to compile `factories.yml`, which is one of the most important configuration file in symfony. This config handler is very particular since it converts a YAML configuration file into the PHP code that ultimately instantiate the factories (the all-important components we talked about earlier). Not your average config handler, is it?

# The Dispatching and Execution of the Request

Enough said about factories, let's get back on track with the dispatch process. Once `sfContext` is finished initializing, the final step is to call the controller's `dispatch()` method, `sfFrontWebController::dispatch()`[92].

The dispatch process itself in symfony is very simple. In fact, `sfFrontWebController::dispatch()` simply pulls the module and action names from the request parameters and forwards the application via `sfController::forward()`[93].

At this point, if the routing could not parse any module name or action name from the current url, an `sfError404Exception`[94] is raised, which will forward the request to the error 404 handling module (see `sf_error_404_module` and `sf_error_404_action`). Note that you can raise such an exception from anywhere in your application to achieve this effect.

The `forward` method is responsible for a lot of pre-execution checks as well as preparing the configuration and data for the action to be executed.

First the controller checks for the presence of a `generator.yml`[95] file for the current module. This check is performed first (after some basic module / action name cleanup) because the `generator.yml` config file (if it exists) is responsible for generating the base actions class for the module (through its config handler, `sfGeneratorConfigHandler`[96]). This is needed for the next step, which checks if the module and action exists. This is delegated to the controller, through `sfController::actionExists()`[97], which in turn

90. http://trac.symfony-project.org/browser/branches/1.3/lib/config/sfDefineEnvironmentConfigHandler.class.php
91. http://trac.symfony-project.org/browser/branches/1.3/lib/config/sfFactoryConfigHandler.class.php
92. http://www.symfony-project.org/api/1_3/sfFrontWebController#method_dispatch
93. http://www.symfony-project.org/api/1_3/sfController#method_forward
94. http://trac.symfony-project.org/browser/branches/1.3/lib/exception/sfError404Exception.class.php
95. http://trac.symfony-project.org/browser/branches/1.3/lib/config/config/generator.yml
96. http://trac.symfony-project.org/browser/branches/1.3/lib/config/sfGeneratorConfigHandler.class.php
97. http://www.symfony-project.org/api/1_3/sfController#method_actionexists

calls the `sfController::controllerExists()`[98] method. Here again, if the `actionExists()` method fails, an `sfError404Exception` is raised.

The `sfGeneratorConfigHandler`[99] is a special config handler that takes care of instantiating the right generator class for your module and executing it. For more information about config handlers, see *A word on config handler* in this chapter. Also, for more information about the `generator.yml`, see chapter 6 of the symfony Reference Book[100].

There's not much you can do here besides overriding the `sfApplicationConfiguration::getControllerDirs()`[101] method in the application's configuration class. This method returns an array of directories where the controller files live, with an additional parameter to tell symfony if it should check whether controllers in each directory are enabled via the `sf_enabled_modules` configuration option from `settings.yml`. For example, `getControllerDirs()` could look something like this:

*Listing 10-5*

```
/**
 * Controllers in /tmp/myControllers won't need to be enabled
 * to be detected
 */
public function getControllerDirs($moduleName)
{
  return array_merge(parent::getControllerDirs($moduleName), array(
    '/tmp/myControllers/'.$moduleName => false
  ));
}
```

If the action does not exist, an `sfError404Exception` is thrown.

The next step is to retrieve an instance of the controller containing the action. This is handled via the `sfController::getAction()`[102] method, which, like `actionExists()` is a facade for the `sfController::getController()`[103], method. Finally, the controller instance is added to the `action stack`.

The action stack is a FIFO (First In First Out) style stack which holds all actions executed during the request. Each item within the stack is wrapped in an `sfActionStackEntry` object. You can always access the stack with `sfContext::getInstance()->getActionStack()` or `$this->getController()->getActionStack()` from within an action.

After a little more configuration loading, we'll be ready to execute our action. The module-specific configuration must still be loaded, which can be found in two distinct places. First symfony looks for the `module.yml` file (normally located in `apps/frontend/modules/`

98. http://www.symfony-project.org/api/1_3/sfController#method_controllerexists
99. http://trac.symfony-project.org/browser/branches/1.3/lib/config/sfGeneratorConfigHandler.class.php
100. http://www.symfony-project.org/reference/1_3/en/06-Admin-Generator
101. http://www.symfony-project.org/api/1_3/sfApplicationConfiguration#method_getcontrollerdirs
102. http://www.symfony-project.org/api/1_3/sfController#method_getaction
103. http://www.symfony-project.org/api/1_3/sfController#method_getcontroller

`yourModule/config/module.yml`) which, because it's a YAML config file, uses the config cache. Additionally, this configuration file can declare the module as *internal*, using the `mod_yourModule_is_internal` setting which will cause the request to fail at this point since an internal module cannot be called publicly.

Internal modules were formerly used to generate email content (through `getPresentationFor()`, for example). You should now use other techniques, such as partial rendering (`$this->renderPartial()`) instead.

Now that `module.yml` is loaded, it's time to check for a second time that the current module is enabled. Indeed, you can set the `mod_$moduleName_enabled` setting to `false` if you want to disable the module at this point.

As mentioned, there are two different ways of enabling or disabling a module. The difference is what happens when the module is disabled. In the first case, when the `sf_enabled_modules` setting is checked, a disabled module will cause an `sfConfigurationException`[104] to be thrown. This should be used when disabling a module permanently. In the second case, via the `mod_$moduleName_enabled` setting, a disabled module will cause the application to forward to the disabled module (see the `sf_module_disabled_module` and `sf_module_disabled_action` settings). You should use this when you want to temporarily disable a module.

The final opportunity to configure a module lies in the `config.php` file (`apps/frontend/modules/yourModule/config/config.php`) where you can place arbitrary PHP code to be run in the context of the `sfController::forward()`[105] method (that is, you have access to the `sfController` instance via the `$this` variable, as the code is literally run inside the `sfController` class).

### The Dispatching Process Summary

1. `sfFrontWebController::dispatch()`[106] is called
2. `sfController::forward()`[107] is called
3. Check for a `generator.yml`
4. Check if the module / action exists
5. Retrieve a list of controllers directories
6. Retrieve an instance of the action
7. Load module configuration through `module.yml` and/or `config.php`

## The Filter Chain

Now that all the configuration has been done, it's time to start the real work. Real work, in this particular case, is the execution of the filter chain.

Symfony's filter chain implements a design pattern known as chain of responsibility[108]. This is a simple yet powerful pattern that allows for chained actions, where each part of the

104. `http://trac.symfony-project.org/browser/branches/1.3/lib/exception/sfConfigurationException.class.php`
105. `http://www.symfony-project.org/api/1_3/sfController#method_forward`
106. `http://www.symfony-project.org/api/1_3/sfFrontWebController#method_dispatch`
107. `http://www.symfony-project.org/api/1_3/sfController#method_forward`
108. `http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern`

chain is able to decide whether or not the chain should continue execution. Each part of the chain is also able to execute both before and after the rest of the chain's execution.

The configuration of the filter chain is pulled from the current module's `filters.yml`[109], which is why the action instance is needed. This is your chance to modify the set of filters executed by the chain. Just remember that the rendering filter should always be the first in the list (we will see why later). The default filters configuration is as follow:

- `rendering`[110]
- `security`[111]
- `cache`[112]
- `execution`[113]

It is strongly advised that you add your own filters between the `security` and the `cache` filter.

## The Security Filter

Since the `rendering` filter waits for everyone to be done before doing anything, the first filter that actually gets executed is the `security` filter. This filter ensures that everything is right according to the `security.yml`[114] configuration file. Specifically, the filter forwards an unauthenticated user to the `login` module / action and a user with insufficient credentials to the `secure` module / action. Note that this filter is only executed if security is enabled for the given action.

## The Cache Filter

Next comes the `cache` filter. This filter takes advantage of its ability to prevent further filters from being executed. Indeed, if the cache is activated, and if we have a hit, why even bother executing the action? Of course, this will work only for a fully cacheable page, which is not the case for the vast majority of pages.

But this filter has a second bit of logic that gets executed after the execution filter, and just before the rendering filter. This code is responsible for setting up the right HTTP cache headers, and placing the page into the cache if necessary, thanks to the `sfViewCacheManager::setPageCache()`[115] method.

109. `http://trac.symfony-project.org/browser/branches/1.3/lib/config/config/filters.yml`
110. `http://trac.symfony-project.org/browser/branches/1.3/lib/filter/sfRenderingFilter.class.php`
111. `http://trac.symfony-project.org/browser/branches/1.3/lib/filter/sfSecurityFilter.class.php`
112. `http://trac.symfony-project.org/browser/branches/1.3/lib/filter/sfCacheFilter.class.php`
113. `http://trac.symfony-project.org/browser/branches/1.3/lib/filter/sfExecutionFilter.class.php`
114. `http://trac.symfony-project.org/browser/branches/1.3/lib/config/config/security.yml`
115. `http://www.symfony-project.org/api/1_3/sfViewCacheManager#method_setpagecache`

## The Execution Filter

Last but not least, the `execution` filter will, finally, take care of executing your business logic and handling the associated view.

Everything starts when the filter checks the cache for the current action. Of course, if we have something in the cache, the actual action execution is skipped and the `Success` view is then executed.

If the action is not found in the cache, then it is time to execute the `preExecute()` logic of the controller, and finally to execute the action itself. This is accomplished by the action instance via a call to `sfActions::execute()`[116]. This method doesn't do much: it simply checks that the action is callable, then calls it. Back in the filter, the `postExecute()` logic of the action is now executed.

The return value of your action is very important, since it will determine what view will get executed. By default, if no return value is found, `sfView::SUCCESS` is assumed (which translates to, you guessed it, `Success`, as in `indexSuccess.php`).

One more step ahead, and it's view time. The filter checks for two special return values that your action may have returned, `sfView::HEADER_ONLY` and `sfView::NONE`. Each does exactly what their names say: sends HTTP headers only (internally handled via `sfWebResponse::setHeaderOnly()`[117]) or skips rendering altogether.

Built-in view names are: `ALERT`, `ERROR`, `INPUT`, `NONE` and `SUCCESS`. But you can basically return anything you want.

Once we know that we *do* want to render something, we're ready to get into the final step of the filter: the actual view execution.

The first thing we do is retrieve an `sfView`[118] object through the `sfController::getView()`[119] method. This object can come from two different places. First you could have a custom view object for this specific action (assuming the current module/action is, let's keep it simple, module/action) `actionSuccessView` or `module_actionSuccessView` in a file called `apps/frontend/modules/module/view/actionSuccessView.class.php`. Otherwise, the class defined in the `mod_module_view_class` configuration entry will be used. This value defaults to `sfPHPView`[120].

Using your own view class gives you a chance to run some view specific logic, through the `sfView::execute()`[121] method. For example, you could instantiate your own template engine.

There are three rendering modes possible for rendering the view:

1. `sfView::RENDER_NONE`": equivalent to `sfView::NONE`, this cancels any rendering from being actually, well, rendered.

116. `http://www.symfony-project.org/api/1_3/sfActions#method_execute`
117. `http://www.symfony-project.org/api/1_3/sfWebResponse#method_setheaderonly`
118. `http://trac.symfony-project.org/browser/branches/1.3/lib/view/sfView.class.php`
119. `http://www.symfony-project.org/api/1_3/sfController#method_getview`
120. `http://trac.symfony-project.org/browser/branches/1.3/lib/view/sfPHPView.class.php`
121. `http://www.symfony-project.org/api/1_3/sfView#method_execute`

2. `sfView::RENDER_VAR`: populates the action's presentation, which is then accessible through its stack entry's `sfActionStackEntry::getPresentation()`[122] method.
3. `sfView::RENDER_CLIENT`, the default mode, will render the view and feed the response's content.

Indeed, the rendering mode is used only through the `sfController::getPresentationFor()`[123] method that returns the rendering for a given module / action

### The Rendering Filter

We're almost done now, just one very last step. The filter chain has almost finished executing, but do you remember the rendering filter? It's been waiting since the beginning of the chain for everyone to complete their work so that it can do its own job. Namely, the rendering filter sends the response content to the browser, using `sfWebResponse::send()`[124].

### Summary of the filter chain execution

1. The filter chain is instantiated with configuration from the `filters.yml` file
2. The `security` filter checks for authorizations and credentials
3. The `cache` filter handles the cache for the current page
4. The `execution` filter actually executes the action
5. The `rendering` filter send the response through `sfWebResponse`

## Global Summary

1. Retrieval of the application's configuration
2. Creation of an `sfContext` instance
3. Initialization of `sfContext`
4. Factories loading
5. Events notified:
    1. `request.filter_parameters`
    2. `routing.load_configuration`
    3. `context.load_factories`
6. Global templates parameters added
7. `sfFrontWebController::dispatch()`[125] is called
8. `sfController::forward()`[126] is called
9. Check for a `generator.yml`
10. Check if the module / action exists
11. Retrieve a list of controllers directories
12. Retrieve an instance of the action
13. Load module configuration through `module.yml` and/or `config.php`
14. The filter chain is instantiated with configuration from the `filters.yml` file
15. The `security` filter checks for authorizations and credentials

122. `http://www.symfony-project.org/api/1_3/sfActionStackEntry#method_getpresentation`
123. `http://www.symfony-project.org/api/1_3/sfController#method_getpresentationfor`
124. `http://www.symfony-project.org/api/1_3/sfWebResponse#method_send`
125. `http://www.symfony-project.org/api/1_3/sfFrontWebController#method_dispatch`
126. `http://www.symfony-project.org/api/1_3/sfController#method_forward`

16. The `cache` filter handles the cache for the current page
17. The `execution` filter actually executes the action
18. The `rendering` filter send the response through `sfWebResponse`

## Final Thoughts

That's it! The request has been handled and we're now ready for another one. Of course, we could write an entire book about symfony's internal processes, so this chapter serves only as an overview. You are more than welcome to explore the source by yourself - it is, and always will be, the best way to learn the true mechanics of any framework or library.

Chapter 11

# Windows and symfony

*by Laurent Bonnet*

## Overview

This document is a new step-by-step tutorial covering the installation, deployment and functional test of the symfony framework on Windows Server 2008.

In order to prepare for Internet deployment, the tutorial can be executed in a dedicated server environment, hosted on the Internet.

Of course, it's possible to complete the tutorial on a local server, or a virtual machine at the reader's workstation.

### Reason for a new Tutorial

Currently, there are two sources of information related to Microsoft Internet Information Server (IIS) on the symfony website[127] [128], but they refer to previous versions that have not evolved with newer versions of Microsoft Windows operating systems, especially Windows Server 2008 (released in February, 2008), which includes many changes of interest to PHP developers:

- IIS version 7, the version embedded in Windows Server 2008, was entirely rewritten to a fully modular design.
- IIS 7 has proven to be very reliable, with very few fixes needed from Windows Update since the launch of the product.
- IIS 7 also includes the FastCGI accelerator, a multi-threaded application pool that takes advantage of the native threading model of Windows operating systems.
- The FastCGI implementation of PHP equates to a 5x to 10x performance improvement in execution, without cache, when compared to traditional ISAPI or CGI deployments of PHP on Windows and IIS.
- More recently, Microsoft opened the curtain on a cache accelerator for PHP, which is in Release Candidate status at the time of this writing (2009-11-02).

127. http://trac.symfony-project.org/wiki/symfonyOnIIS
128. http://www.symfony-project.org/cookbook/1_2/en/web_server_iis

**Planned Extension for this Tutorial**

A supplemental section of this chapter is in the works and will be released on the symfony project web site shortly after the publication of this book. It covers the connection to MS SQL Server via PDO, something Microsoft is planning improvements for soon.

*Listing 11-1*

```
[PHP_PDO_MSSQL]
extension=php_pdo_mssql.dll
```

Currently, the best performance in code execution is obtained by the Microsoft native SQL Server driver for PHP 5, an open-source driver available on Windows and currently available in version 1.1. This is implemented as a new PHP extension DLL:

*Listing 11-2*

```
[PHP_SQLSRV]
extension=php_sqlsrv.dll
```

It is possible to use either Microsoft SQL Server 2005 or 2008 for the database. The planned tutorial extension will cover the usage of the edition that is available for free: SQL Server Express.

## How to play with this Tutorial on different Windows Systems, including 32-bit

This document was written specifically for 64-bit editions of Windows Server 2008. However, you should be able to use other versions without any complications.

The exact version of operating software used in the screenshots is Windows Server 2008 Enterprise Edition with Service Pack 2, 64-bit edition.

### 32-bit Versions of Windows

The tutorial is easily portable to 32-bit versions of Windows, by replacing the following references in the text:

- On 64-bit editions: `C:\Program Files (x86)\` and `C:\Windows\SysWOW64\`
- On 32-bit editions: `C:\Program Files\` and `C:\Windows\System32\`

### About Editions other than Enterprise

Also, if you don't have Enterprise Edition, this is not a problem. This document is directly portable to other editions of Windows Server software: Windows Server 2008 Web, Standard or Datacenter Windows Server 2008 Web, Standard or Datacenter with Service Pack 2 Windows Server 2008 R2 Web, Standard, Enterprise or Datacenter editions.

Please note that all editions of Windows Server 2008 R2 are only available as 64-bit operating systems.

### About International Editions

The regional settings used in the screenshots are `en-US`. We also installed an international language package for France.

It is possible to execute the tutorial on Windows client operating systems: Windows XP, Windows Vista and Windows Seven both in x64 and x86 modes.

## Web Server used throughout the Document

The web server used is Microsoft Internet Information Server version 7.0, which is included in all Windows Server 2008 editions as a role. We begin the tutorial with a fully functional Windows Server 2008 server and install IIS from scratch. The install steps use the default choices, we simply add two specific modules that come with the IIS 7.0 modular design: **FastCGI** and **URL Rewrite**.

## Databases

SQLite is the pre-configured database for the symfony sandbox. On Windows, there's nothing specific to install: SQLite support is directly implemented in the PDO PHP extension for SQLite, which is installed at the time of PHP installation.

Hence, there's no need to download and run a separate instance of SQLITE.EXE:

*Listing 11-3*

```
[PHP_PDO_SQLITE]
extension=php_pdo_sqlite.dll
```

## Windows Server Configuration

It is better to use a fresh installation of Windows Server in order to match the step-by-step screenshots in this chapter.

Of course you can work directly on an existing machine, but you may encounter differences due to installed software, runtime, and regional configurations.

In order for you to get the same screenshots as displayed in this tutorial, we recommend obtaining a dedicated Windows Server in a virtual environment, available for free on the Internet for a period of 30 days.

### How to get a free Windows Server Trial?

It is of course possible to use any dedicated server with Internet access. A physical server or even virtual dedicated server (VDS) will work perfectly.

A 30-day server with Windows is available as a trial from Ikoula, a French web host who offers a comprehensive list of services for developers and designers. This trial starts at 0 € / month for a Windows virtual machine running in a Microsoft Hyper-V environment. Yes, you can get a fully functional virtual machine with Windows Server 2008 Web, Standard, Enterprise or even Datacenter edition for FREE for a period of 30 days.

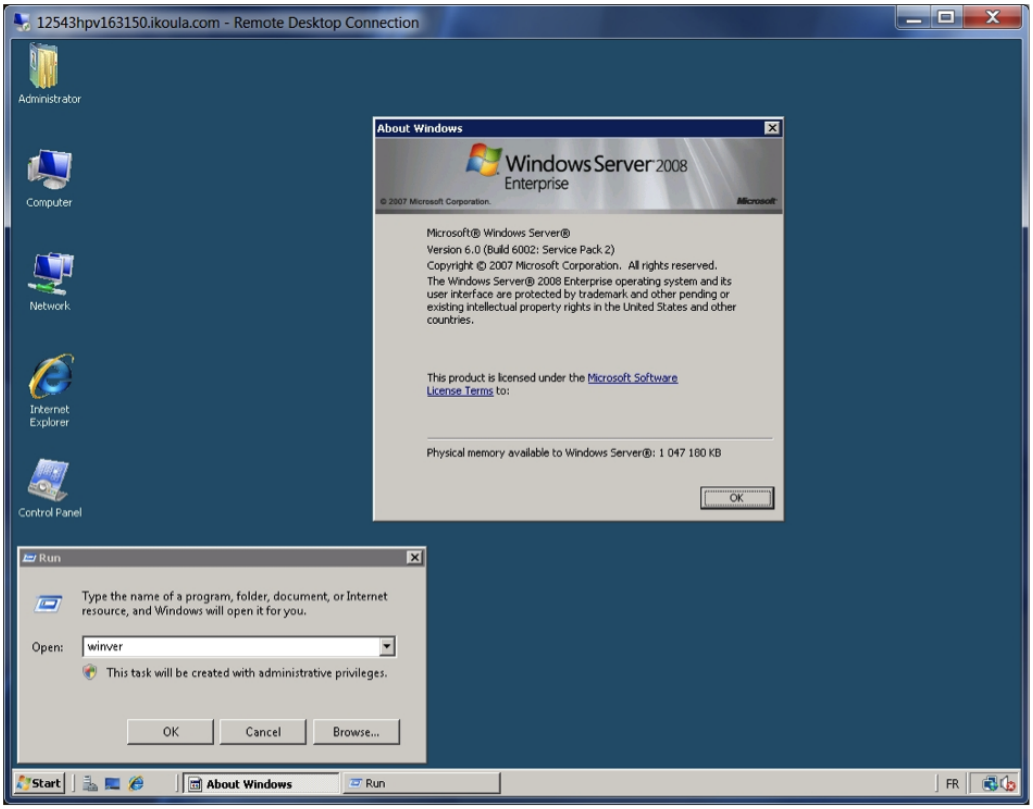
To order, just open your browser to http://www.ikoula.com/flex_server and click on the "Testez gratuitement" button.

In order to get the same messages as outlined in this document, the operating system we ordered with the Flex server is: "Windows Server 2008 Enterprise Edition 64 bits". This is an x64 distribution, delivered with both fr-FR and en-US locales. It's easy to switch from `fr-FR` to `en-US` and vice-versa from the Windows Control Panel. Specifically, this setting can be found in the "Regional and Language Options", which lives under the "Keyboards and Languages" tab. Just click on "Install/uninstall languages".

It is mandatory to have Administrator access to the server.

If working from a remote workstation, the reader should run Remote Desktop Services (formerly known as Terminal Server Client) and ensure he has Administrator access.

The distribution used here is: Windows Server 2008 with Service Pack 2.

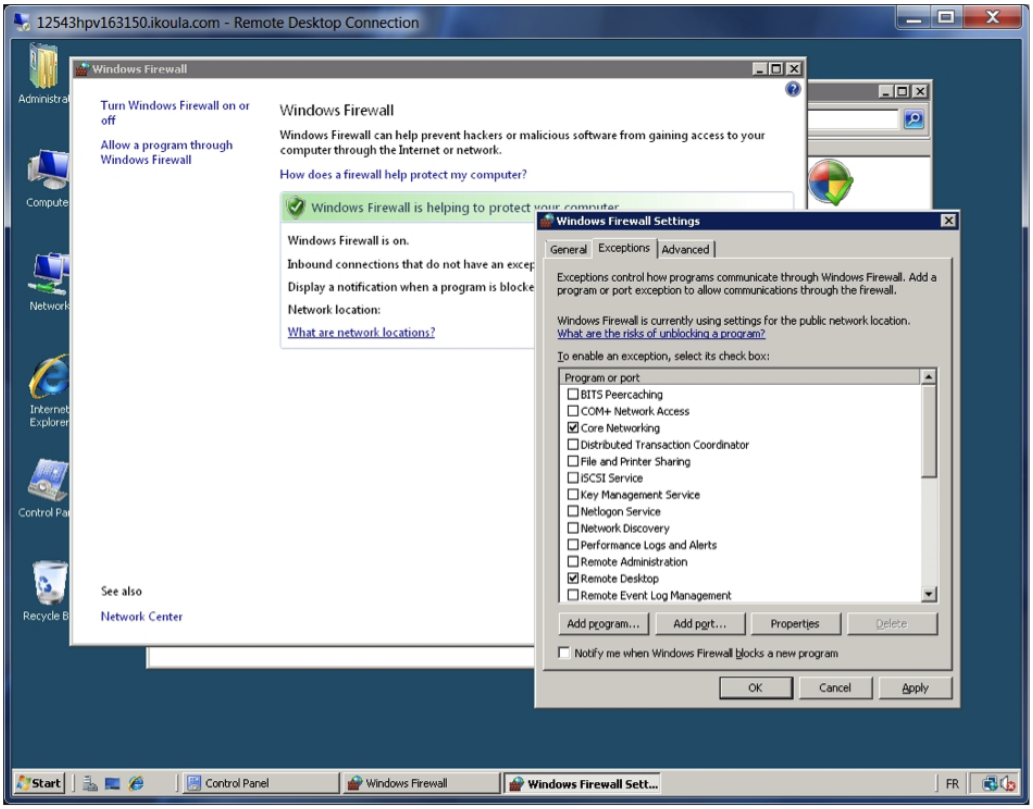



Windows Server 2008 was installed with the graphical environment, which matches Windows Vista's look and feel. It is also possible to use a command-line only for version of Windows Server 2008 with the same services in order to reduce the size of the distribution (1.5 GB instead of 6.5 GB). This also reduces the attack surface and the number of Windows Update patches that will need to be applied.

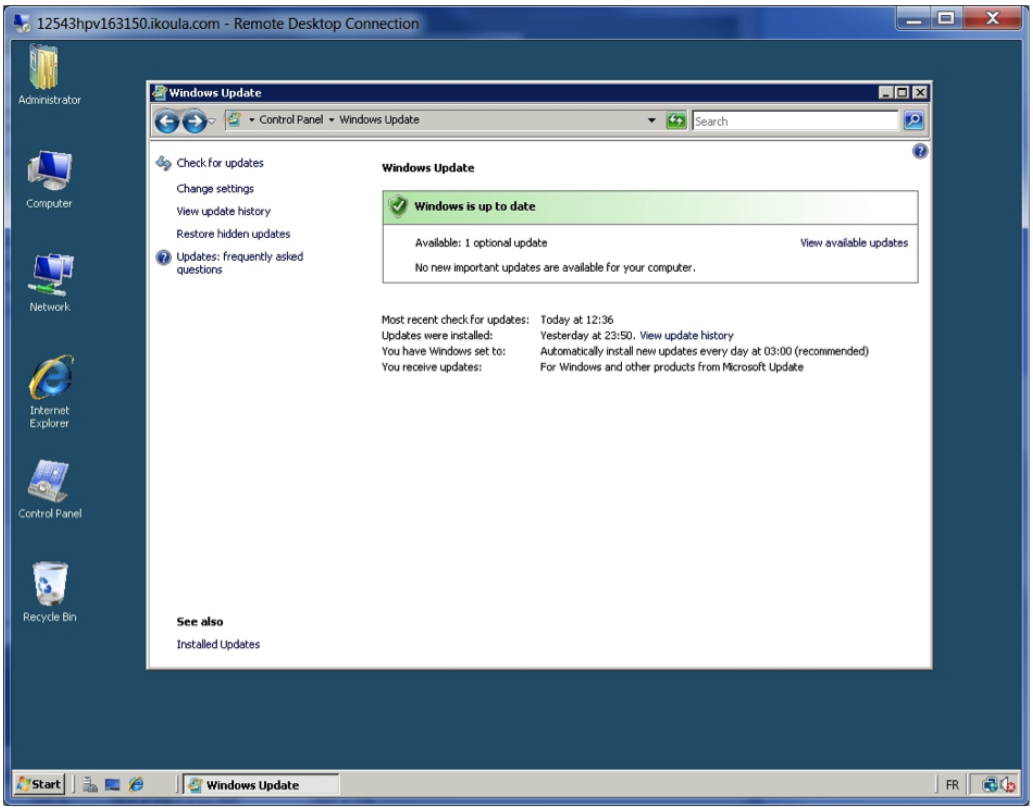
## Preliminary Checks - Dedicated Server on the Internet

Since the server is directly accessible on the Internet, it's always a good idea to check that the Windows Firewall is providing active protection. The only exceptions that should be checked are:

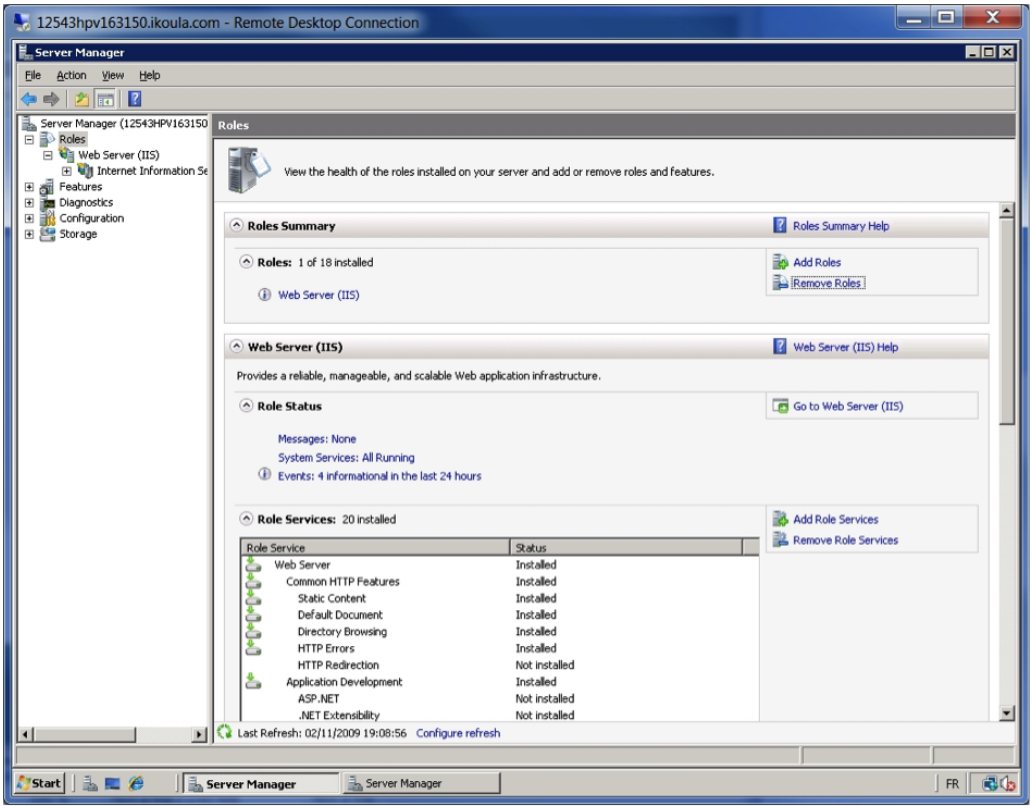
- Core Networking
- Remote Desktop (if accessed remotely)
- Secure World Wide Web Services (HTTPS)
- World Wide Web Services (HTTP)




Then, it's always good to run a round of Windows Update to ensure all software pieces are up-to-date with the latest fixes, patches, and documentation.




As a last step of preparation, and for the sake of removing any potential conflicting parameters in the existing Windows distribution or IIS configuration, we recommend that you uninstall the Web role in Windows server, if previously installed.

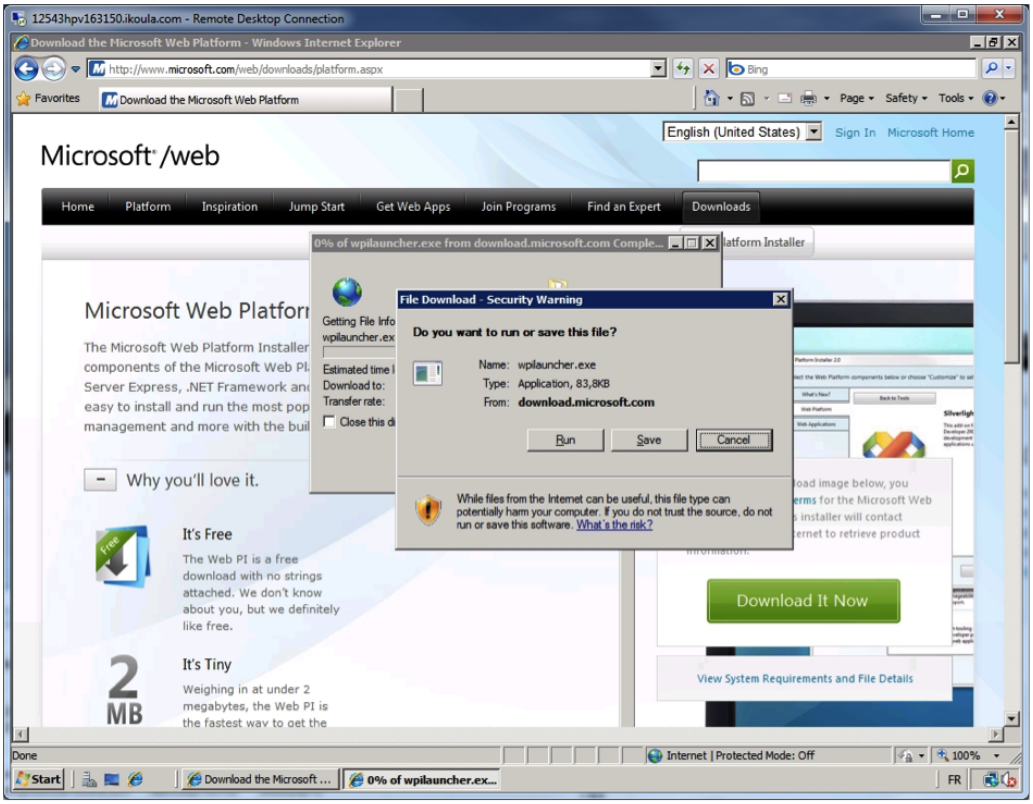



# Installing PHP - Just a few Clicks away

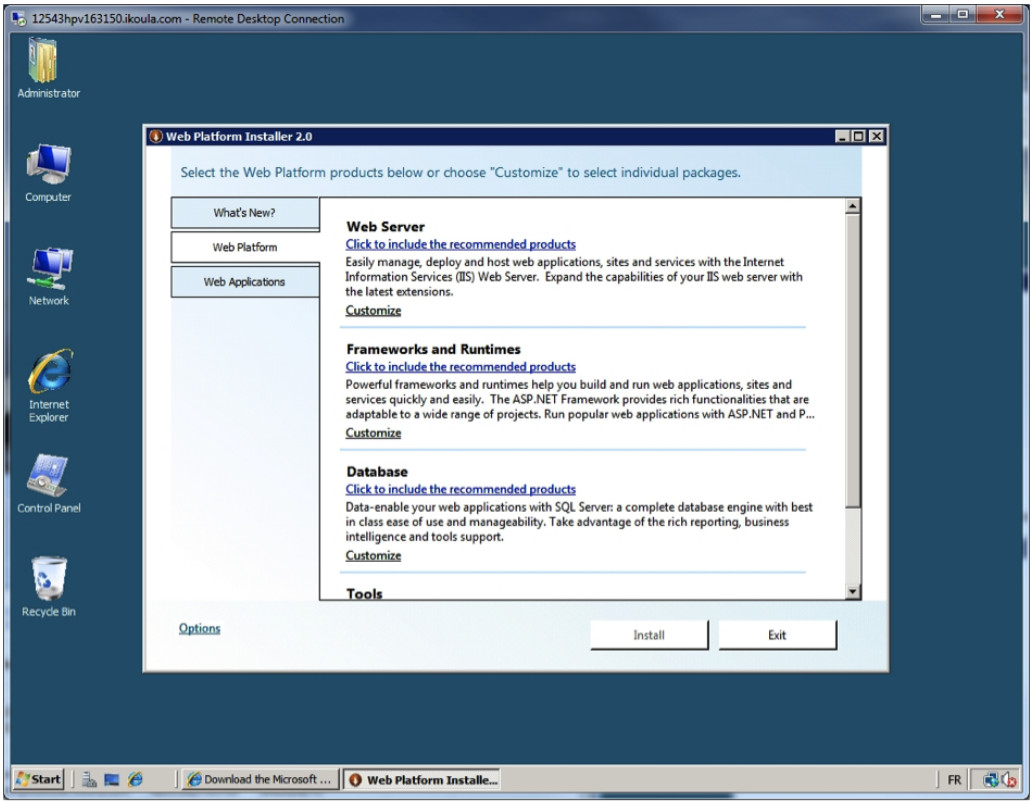
Now, we can install IIS and PHP in one simple operation.

PHP is NOT a part of the Windows Server 2008 distribution, hence we need to first install the Microsoft Web Platform Installer 2.0, referred to as Web PI in the following sections.

Web PI takes care of installing all dependencies necessary for executing PHP on any Windows/IIS system. Hence, it deploys IIS with the minimal Role Services for the Web Server, and also provides minimal options for PHP runtime.

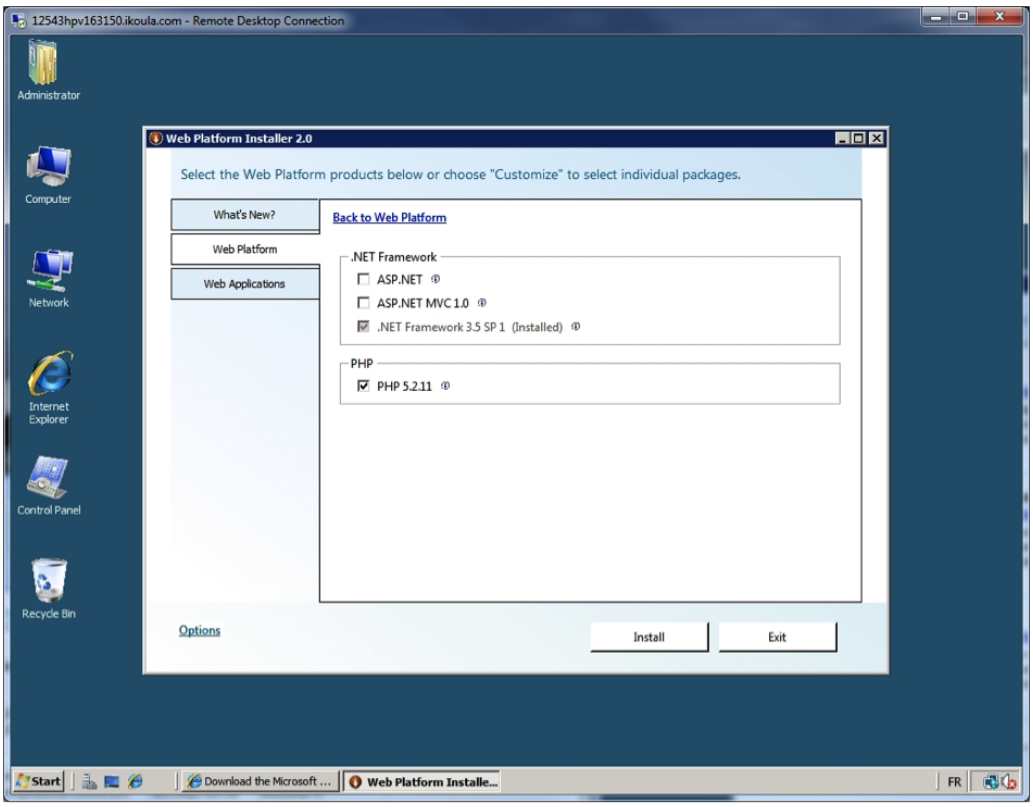



The installation of Microsoft Web Platform Installer 2.0 contains a configuration analyzer, checks for existing modules, proposes any necessary module upgrades, and even allows you to beta-test un-released extensions of the Microsoft Web Platform.

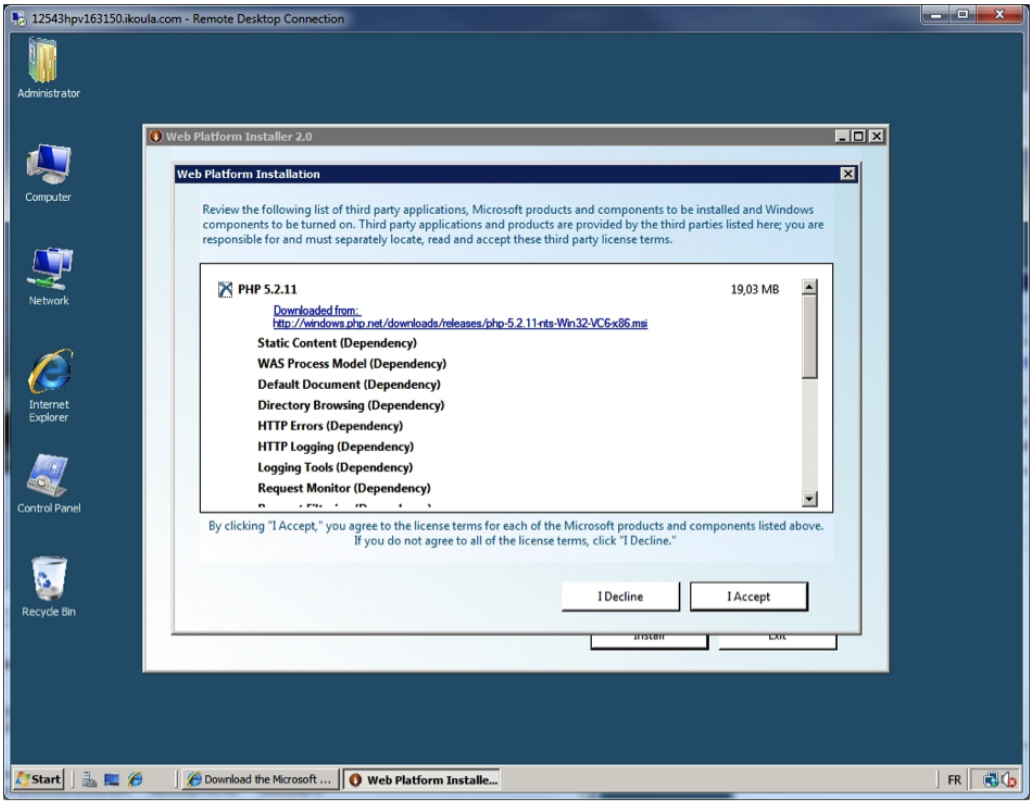


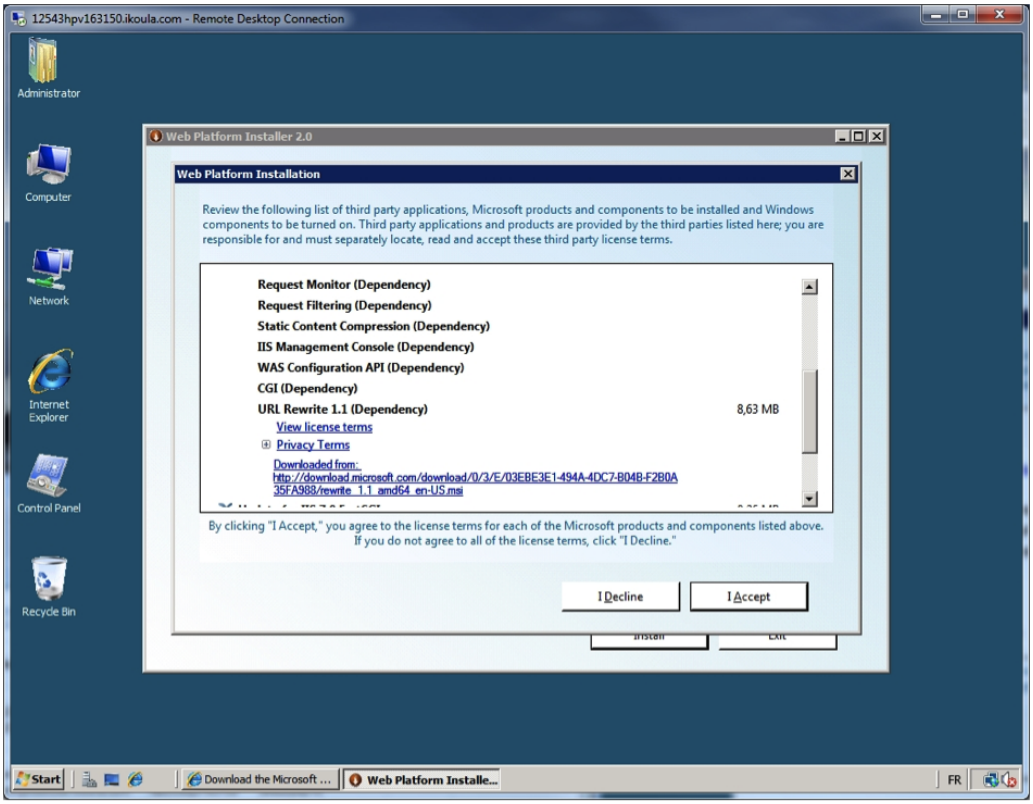

Web PI 2.0 offers PHP runtime installation in one click. The selection installs the “non-thread safe” Win32 implementation of PHP, which is best associated with IIS 7 and FastCGI. It also

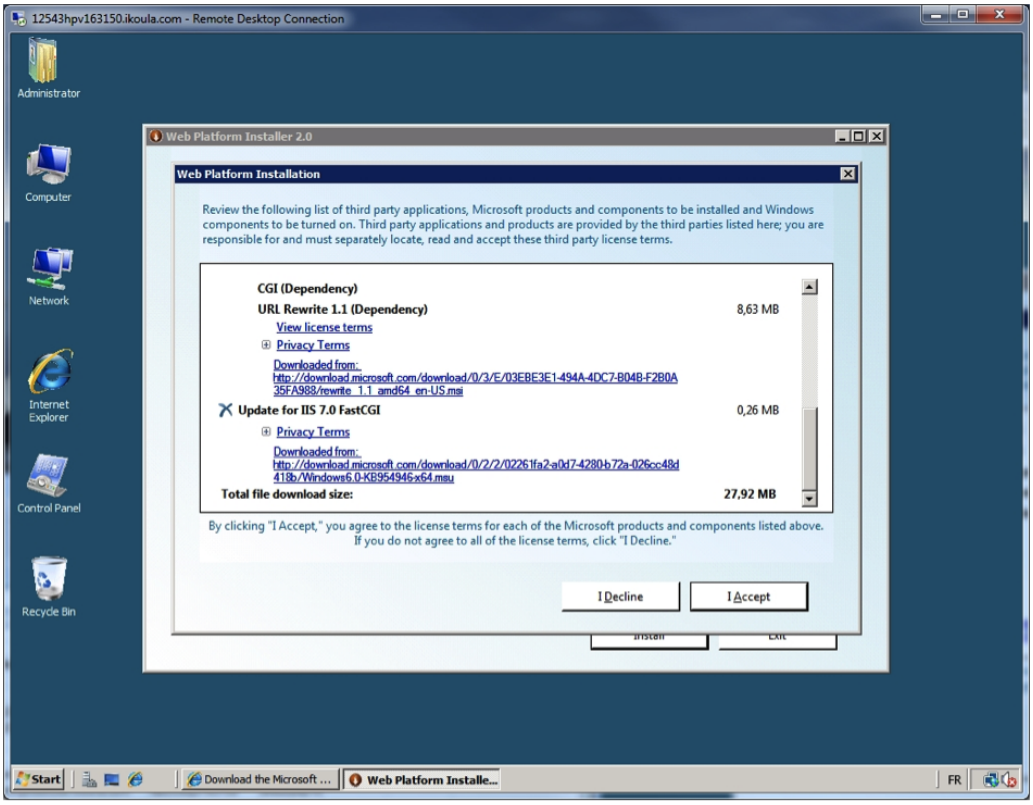
offers the most recently tested runtime, here 5.2.11. To find it, just select the “Frameworks and Runtimes” tab on the left:




After selecting PHP, Web PI 2.0 automatically selects all dependencies needed to service `.php` web pages stored on the server, including the minimal IIS 7.0 roles services:

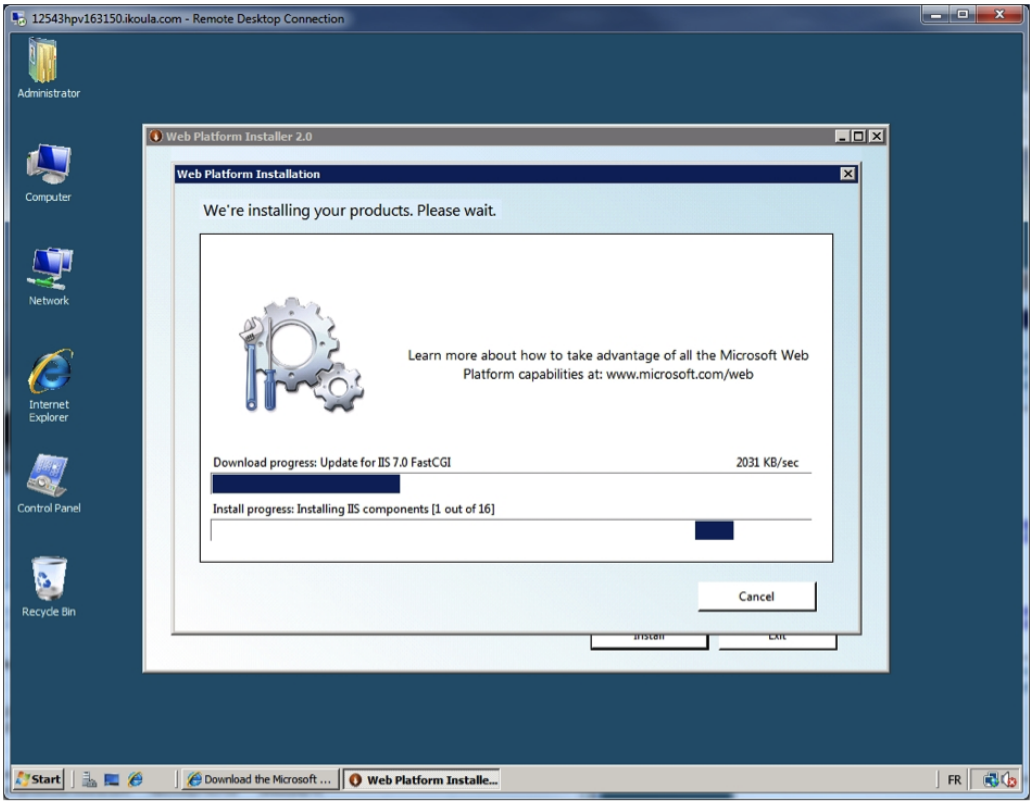




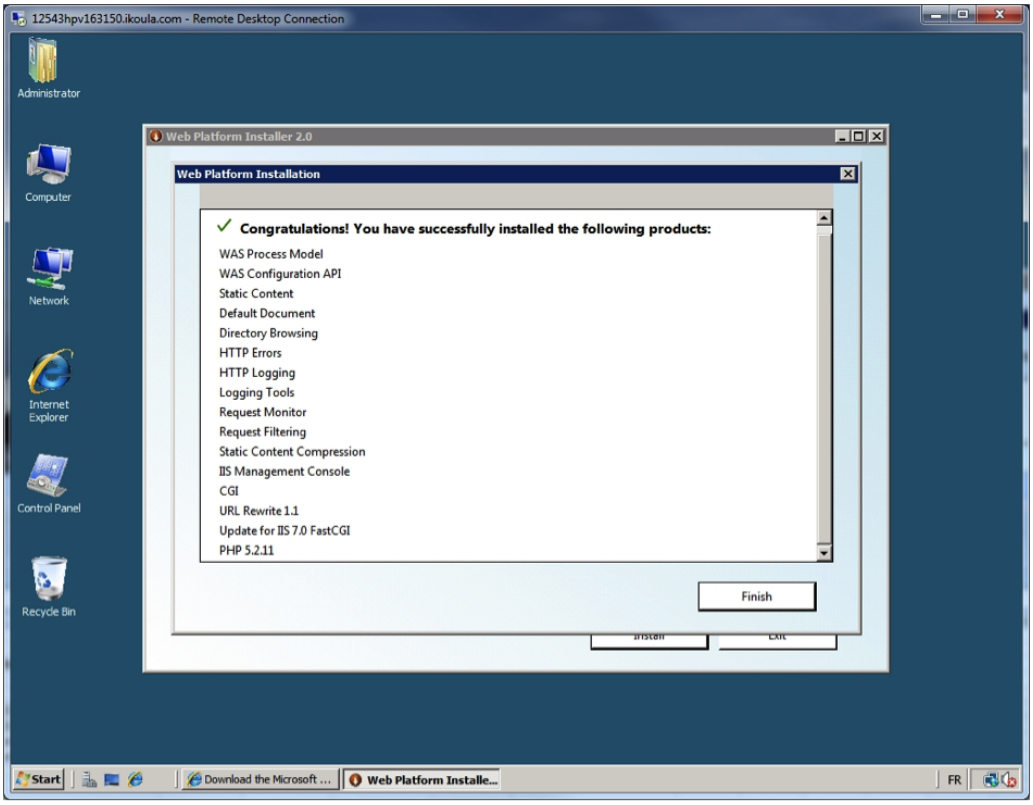




Next, click on Install, then on the “I Accept” button. The installation of IIS components will begin while, in parallel, PHP is downloaded runtime[129] and some module are updated (an update for IIS 7.0 FastCGI for instance).

129. `http://windows.php.net`



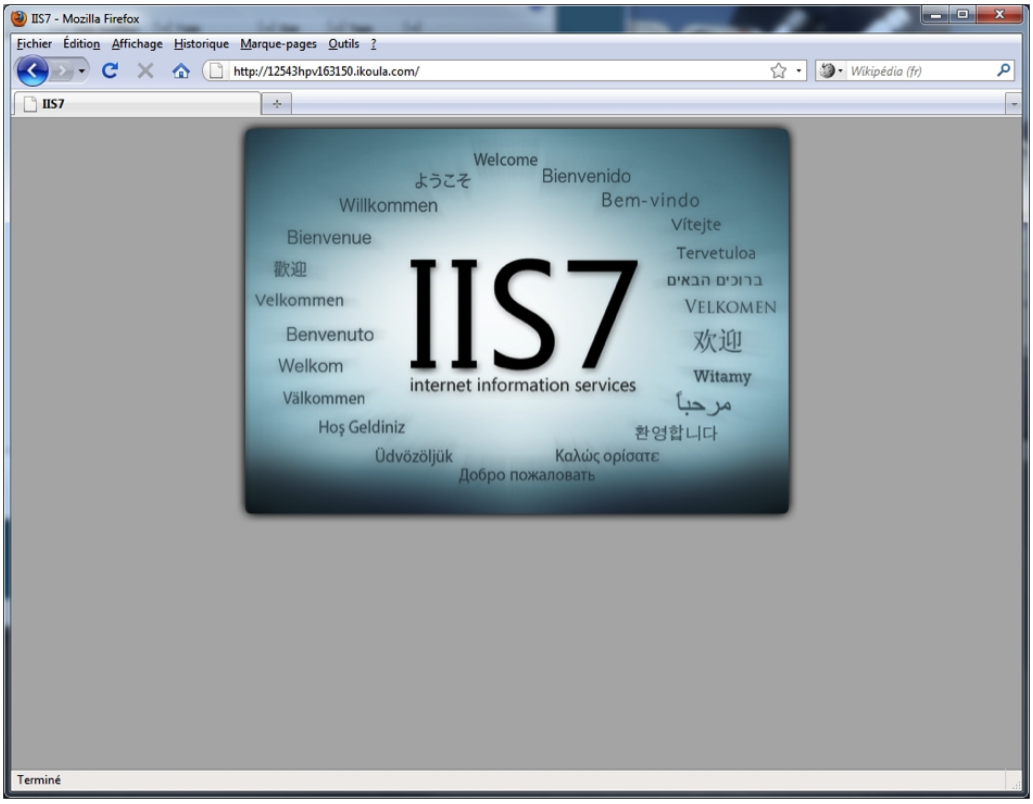

Finally, the PHP setup program executes, and, after a few minutes, should display:




Click on “Finish”.

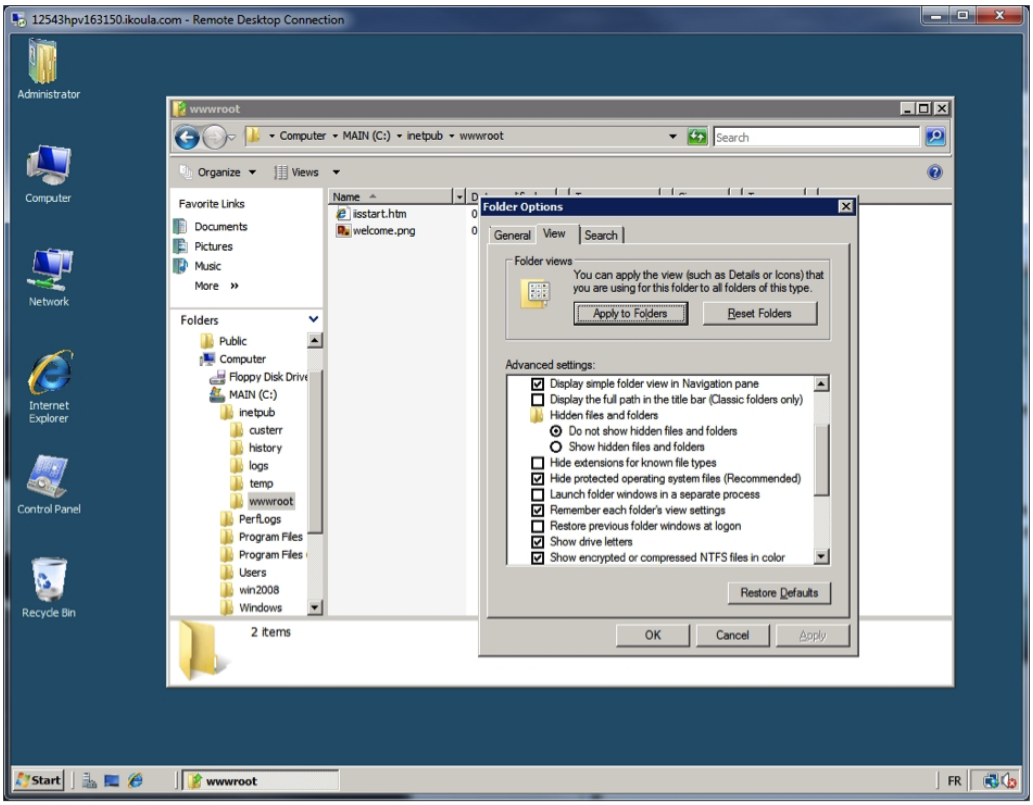
The Windows Server is now listening and able to answer on port 80.

Let’s check this in the browser:

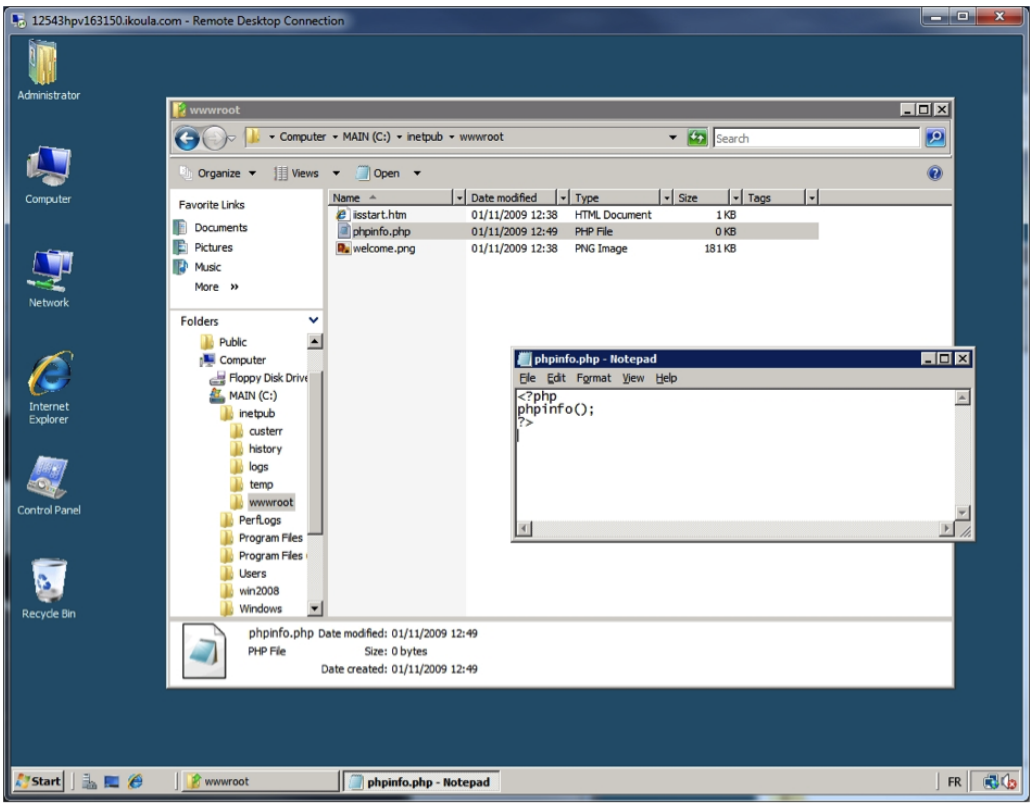



Now, to check that PHP is correctly installed, and available from IIS, we create a small `phpinfo.php` file to be accessed by the default web server on port 80, at `C:\inetpub\wwwroot`.

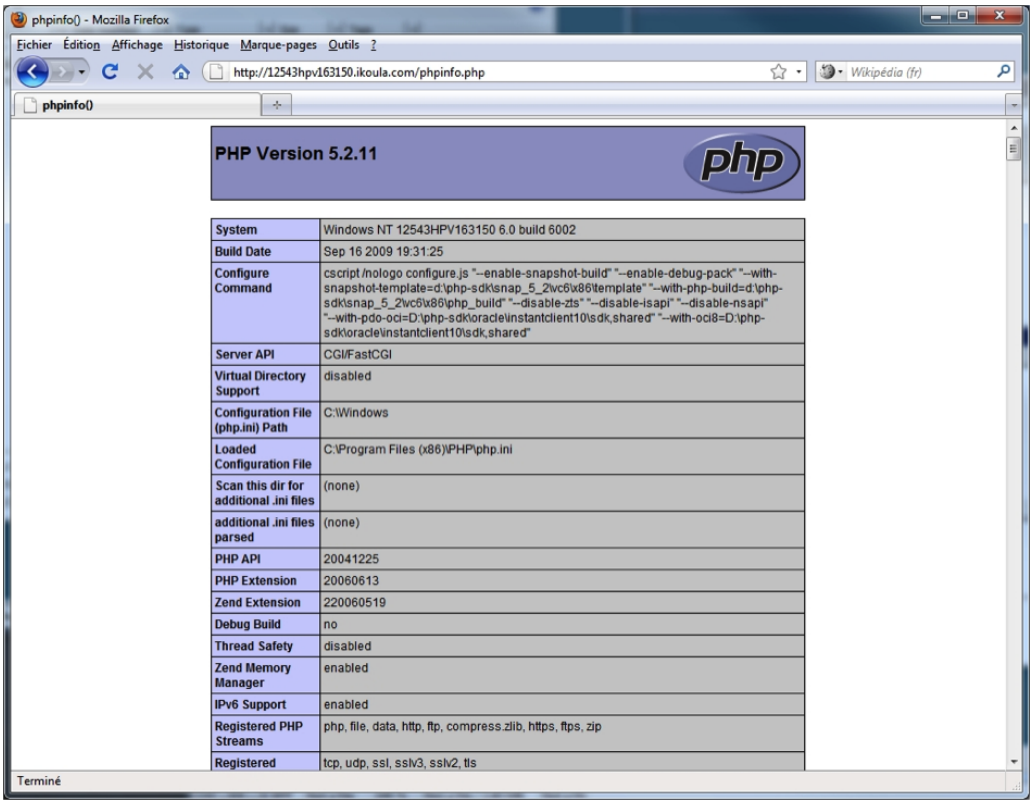
Before doing this, ensure that, in Windows Explorer, we can see the correct extensions of the files. Select “Unhide Extensions for Known Files Types”.



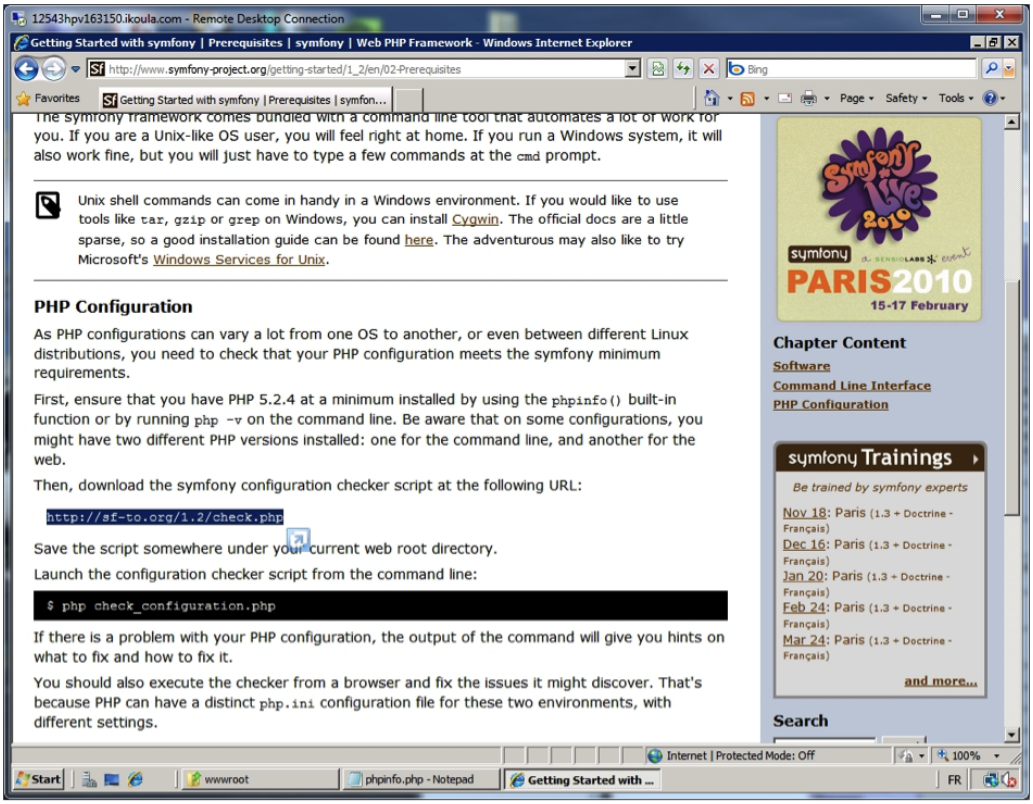
Open Windows Explorer, and go to `C:\inetpub\wwwroot`. Right-click and, select "New Text Document". Rename it to `phpinfo.php` and copy the usual function call:



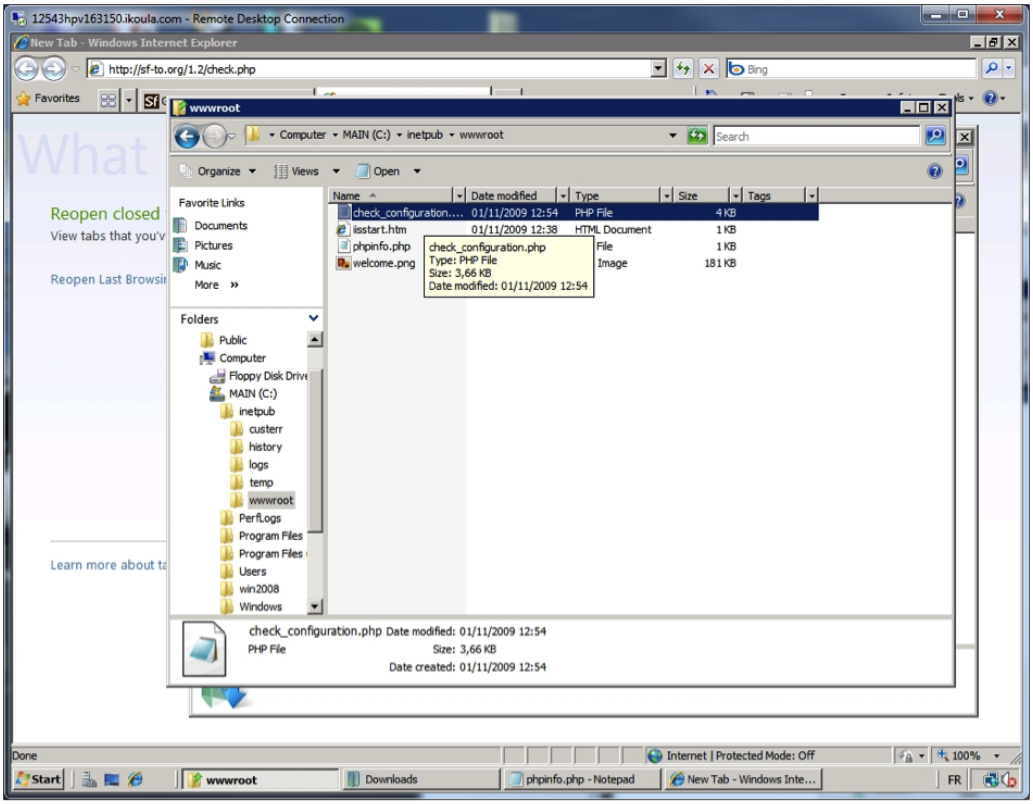

Next, re-open the web browser, and put `/phpinfo.php` at the end of the server's URL:




Finally, to ensure that symfony will install without any problems, download the http://sf-to.org/1.3/check.php[130].

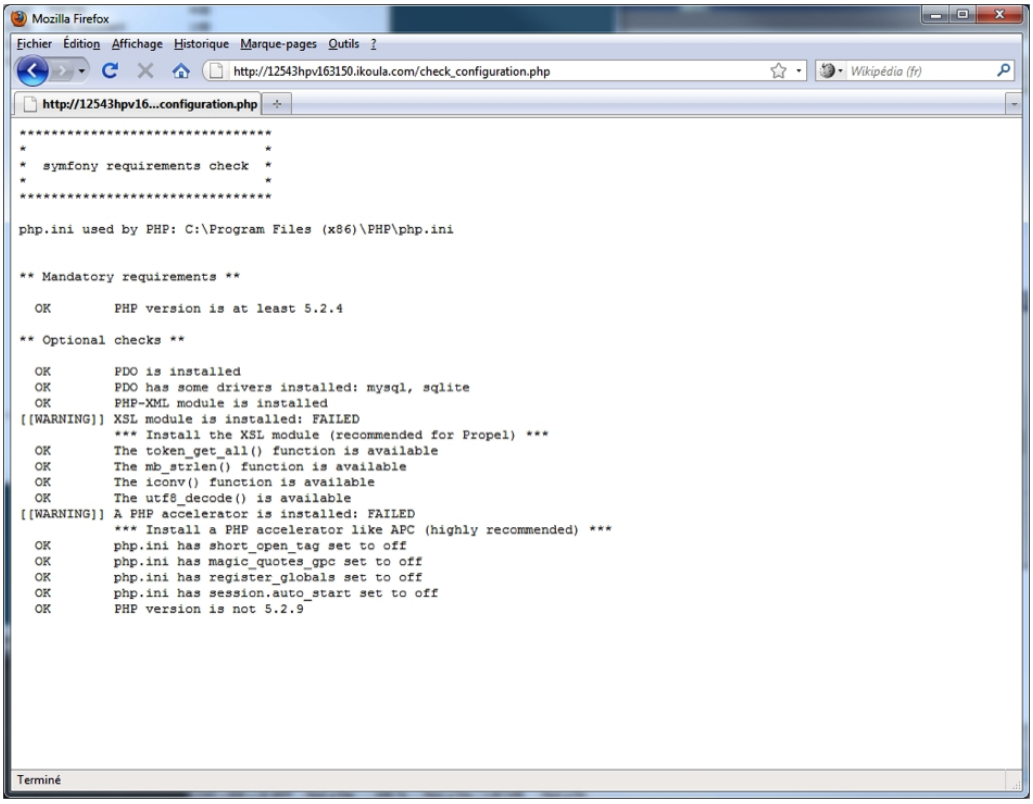



Copy it to the same directory as `phpinfo.php` (`C:\inetpub\wwwroot`) and rename it to `check_configuration.php` if necessary.




Finally, re-open the web browser one last time for now, and put `/check_configuration.php` at the end of the server's URL:

130. `check configuration.php`




# Executing PHP from the Command Line Interface

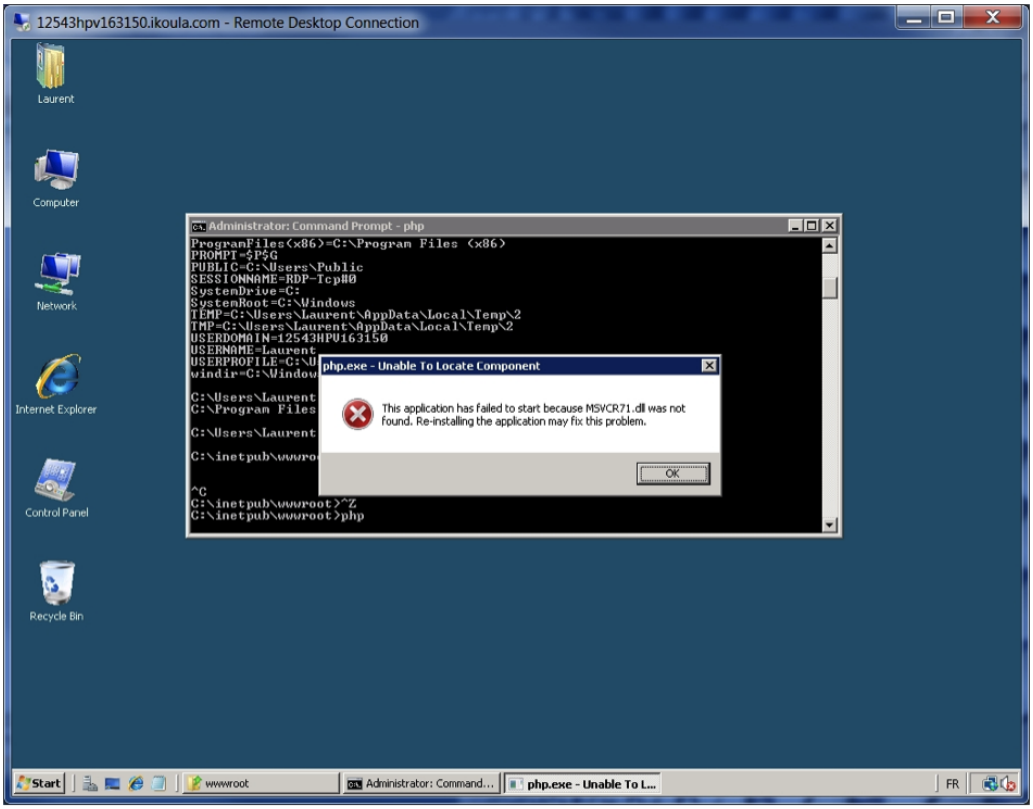
In order to later execute command line tasks with symfony, we need to ensure that PHP.EXE is accessible from the command prompt and executes correctly.

Open a command prompt to `C:\inetpub\wwwroot` and type

```
PHP phpinfo.php
```

*Listing 11-4*

The following error message should appear:




If we don't do anything, the execution of PHP.EXE hangs on the absence of MSVCR71.DLL. So, we must find the DLL file and install it at the correct location.

This `MSVCR71.DLL` is an old version of the Microsoft Visual C++ runtime, which dates back to the 2003 era. It is contained in the .Net Framework 1.1 redistributable package.

The .Net Framework 1.1 redistributable package, can be downloaded at MSDN[131]

The file we're looking for is installed in the following directory: `C:\Windows\Microsoft.NET\Framework\v1.1.4322`

Just copy the `MSVCR71.DLL` file to the following destination:

- on x64 systems: the `C:\windows\syswow64` directory
- on x86 systems: the `C:\windows\system32` directory

We can now uninstall the .Net Framework 1.1.

The PHP.EXE executable can now be run from the command prompt without error. For instance:

*Listing 11-5*

```
PHP phpinfo.php
PHP check_configuration.php
```

Later, we'll verify that SYMFONY.BAT (from the Sandbox distribution) also gives the expected response, which is the syntax of the symfony command.

# Symfony Sandbox Installation and Usage

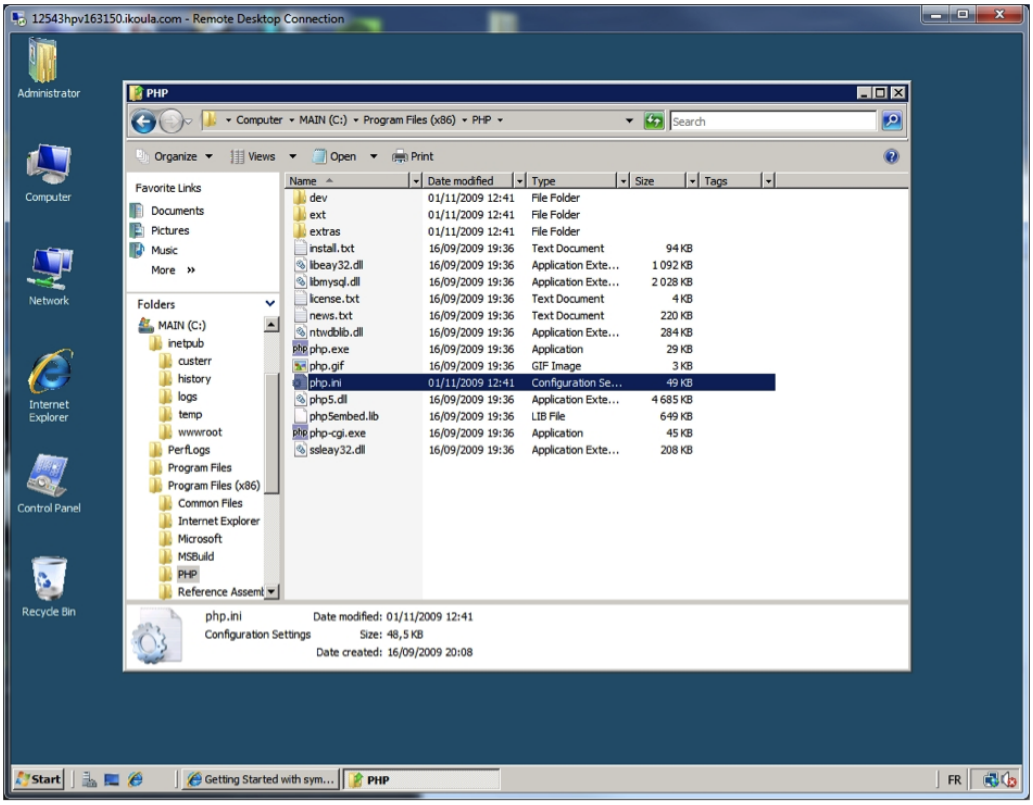
The following paragraph is an excerpt from the "Getting Started with symfony", "The Sandbox"[132]: page: "The sandbox is a dead-easy-to-install pre-packaged symfony project,

131. `http://msdn.microsoft.com/en-us/netframework/aa569264.aspx`
132. `http://www.symfony-project.org/getting-started/1_3/en/A-Sandbox`

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".

The sandbox is pre-configured to use SQLite as a database engine. On Windows, there's nothing specific to install: SQLite support is directly implemented in the PDO PHP extension for SQLite, which is installed at the time of PHP's installation. We already accomplished this earlier when the PHP runtime was installed via the Microsoft Web PI.

Simply check that the SQLite extension is correctly referred to in the PHP.INI file, which resides in the `C:\Program Files (x86)\PHP` directory, and that the DLL implementing PDO support for SQLite is set to `C:\Program Files (x86)\PHP\ext\php_pdo_sqlite.dll`.

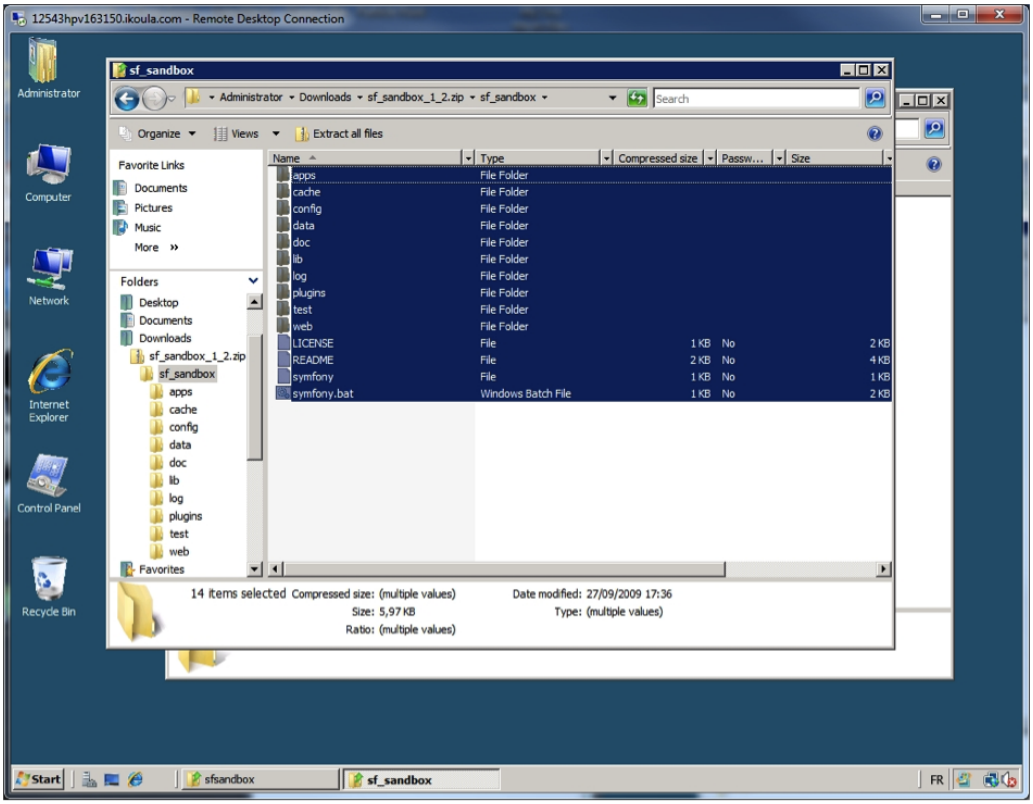



## Download, create Directory, copy all Files

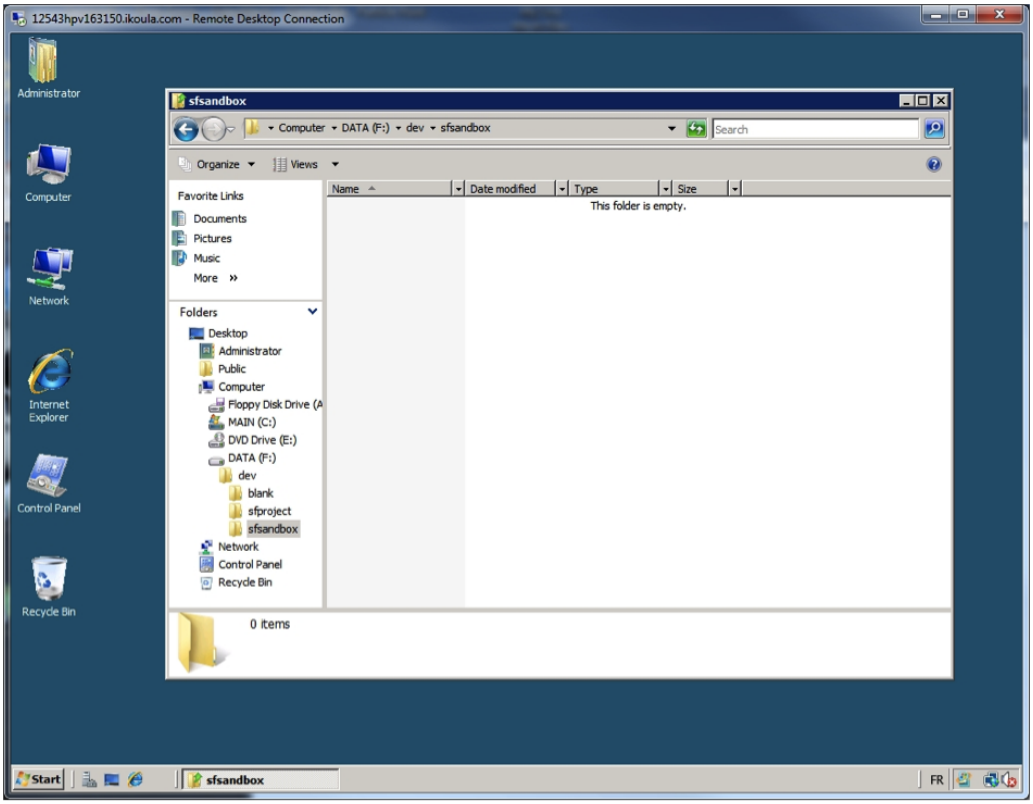
The symfony sandbox project is "ready to install and run", and comes in a `.zip` archive.

Download the archive[133] and extract it to a temporary location, such as the "downloads" directory, which is available for R/W in the `C:\Users\Administrator` directory.

133. `http://www.symfony-project.org/get/sf_sandbox_1_3.zip`

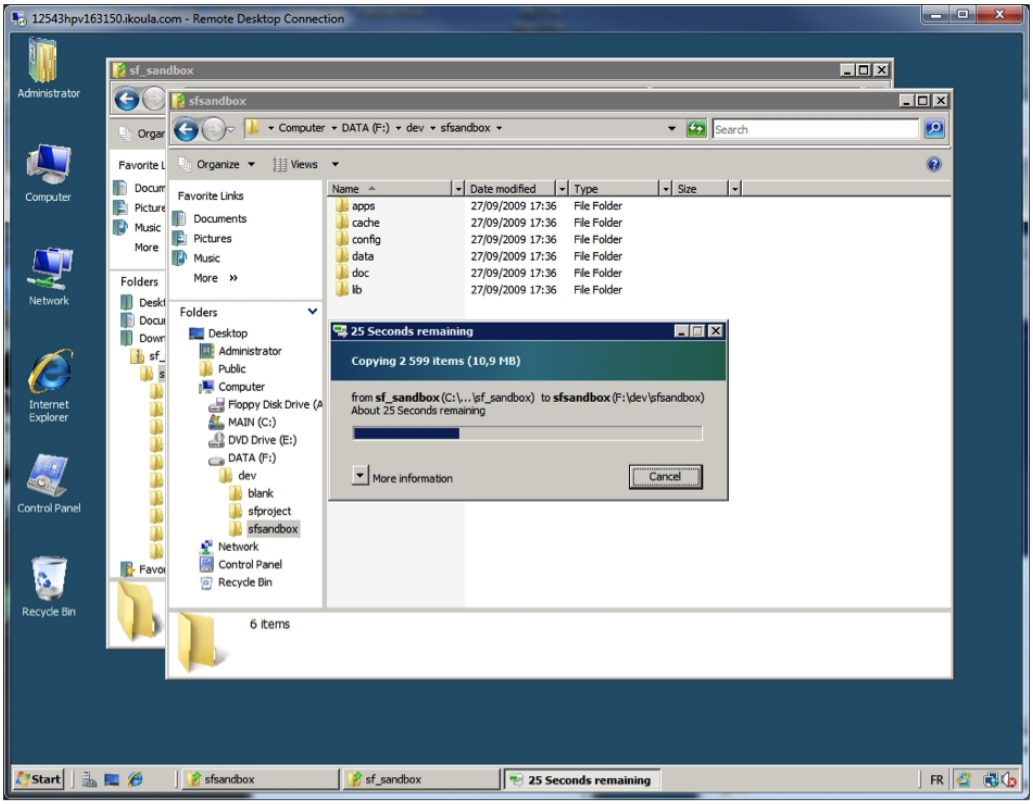



Create a directory for the final destination of the sandbox, such as `F:\dev\sfsandbox`:




Select all files - `CTRL-A` in Windows Explorer - from your download location (source), and copy them to the `F:\dev\sfsandbox` directory.

You should see 2599 items copied to the destination directory:




## Execution Test

Open the command prompt. Change to `F:\dev\sfsandbox` and execute the following command:

```
PHP symfony -V
```

*Listing 11-6*

This should return:

```
symfony version 1.3.0 (F:\dev\sfsandbox\lib\symfony)
```

*Listing 11-7*

From the same command prompt, execute:

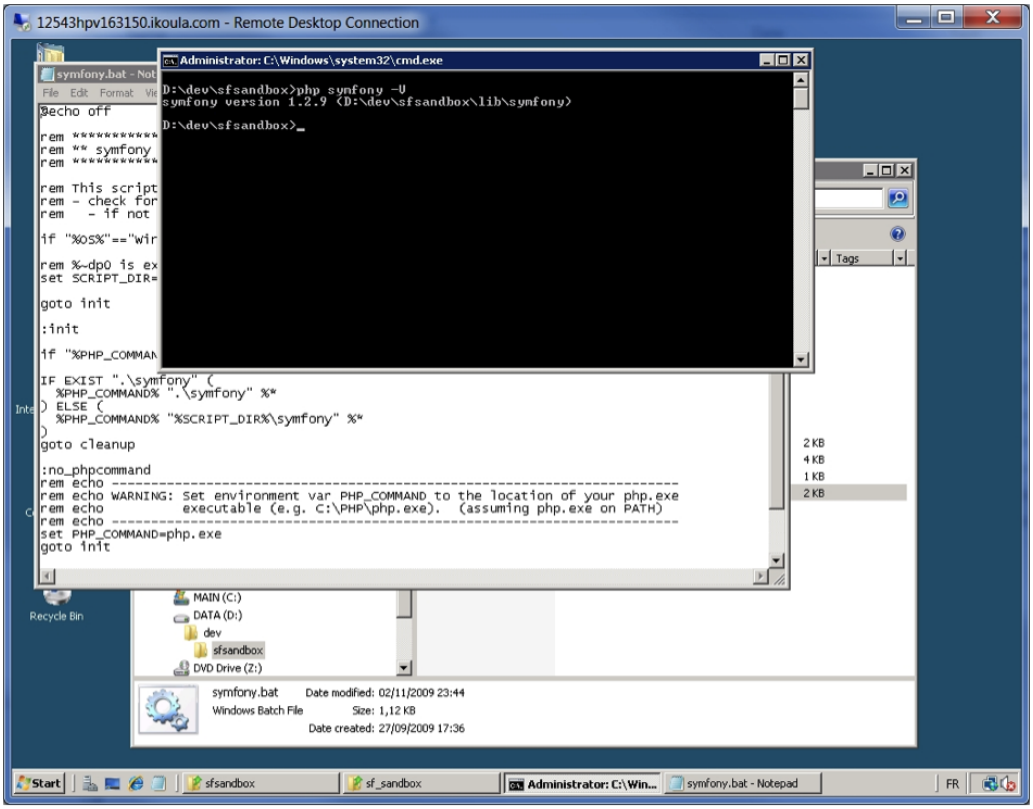
```
SYMFONY.BAT -V
```

*Listing 11-8*

This should return the same result:

```
symfony version 1.3.0 (F:\dev\sfsandbox\lib\symfony)
```

*Listing 11-9*



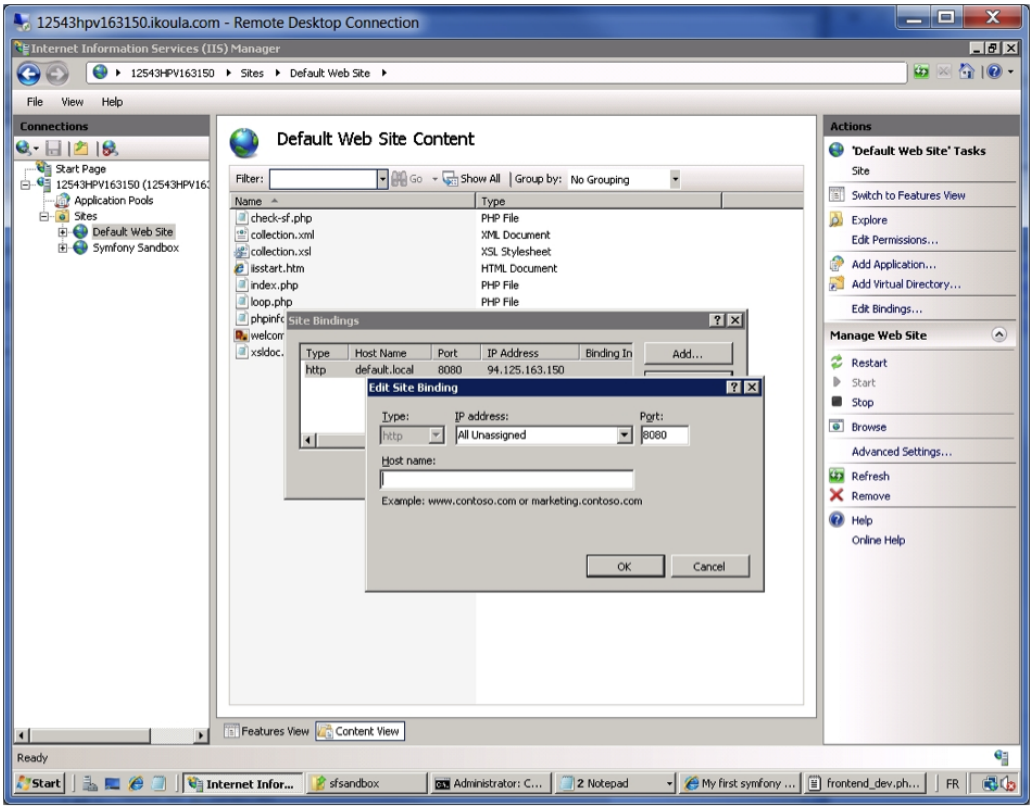

## Web Application Creation

To create a web application on the local server, we use the IIS7 manager, which is the graphical user interface control panel for all IIS related activities. All actions triggered from that UI are actually executed behind-the-scenes via the command line interface.

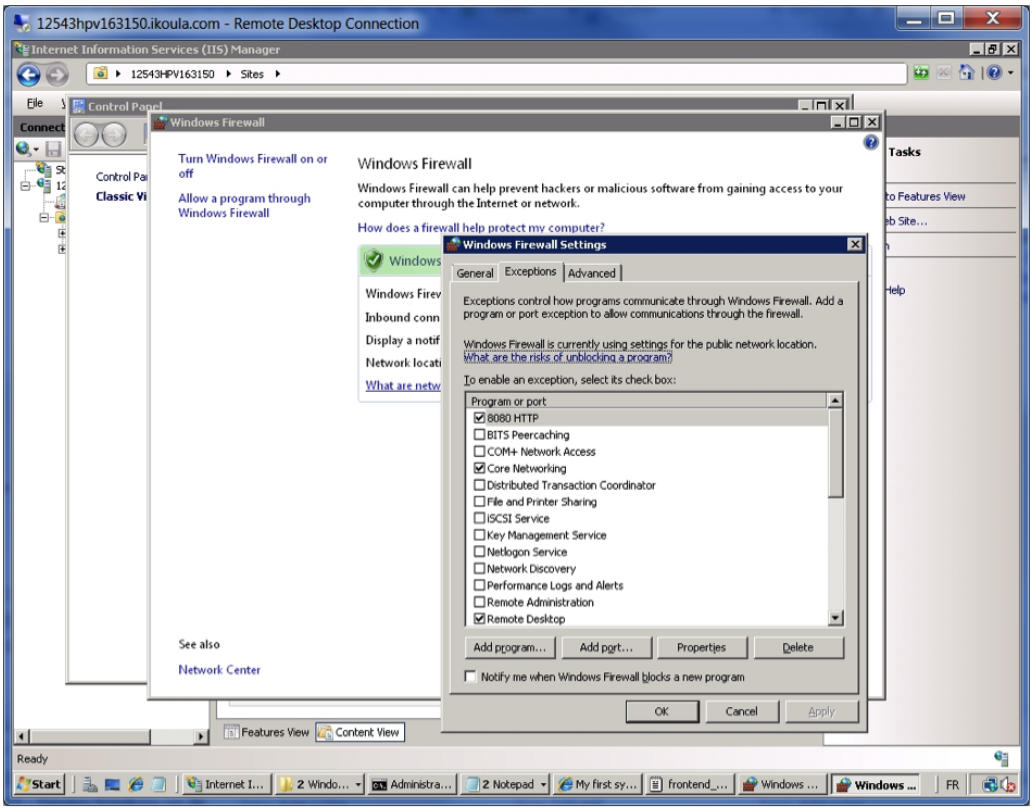
The IIS Manager console is accessible from the Start Menu at Programs, Administrative Tools, Internet Information Server (IIS) Manager.

### Reconfigure “Default Web Site” so it does not interfere on Port 80

We want to ensure that only our symfony sandbox is responding on port 80 (HTTP). To do this, change the existing “Default Web Site” port to 8080.

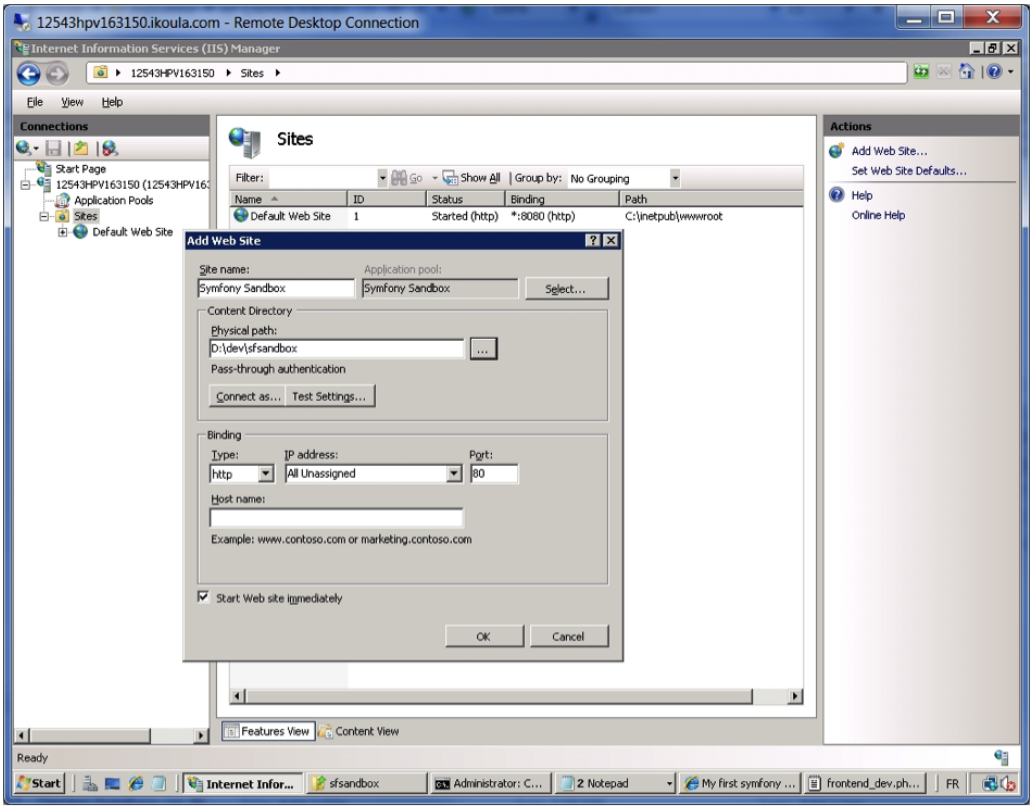



Please note that if Windows Firewall is active, you may have to create an exception for port 8080 to still be able to reach the “Default Web Site”. For that purpose, Go to Windows Control Panel, select Windows Firewall, click on “Allow a program through Windows Firewall”, and click on “Add port” to create that exception. Check the box to enable it after creation.



## Add a new web Site for the Sandbox

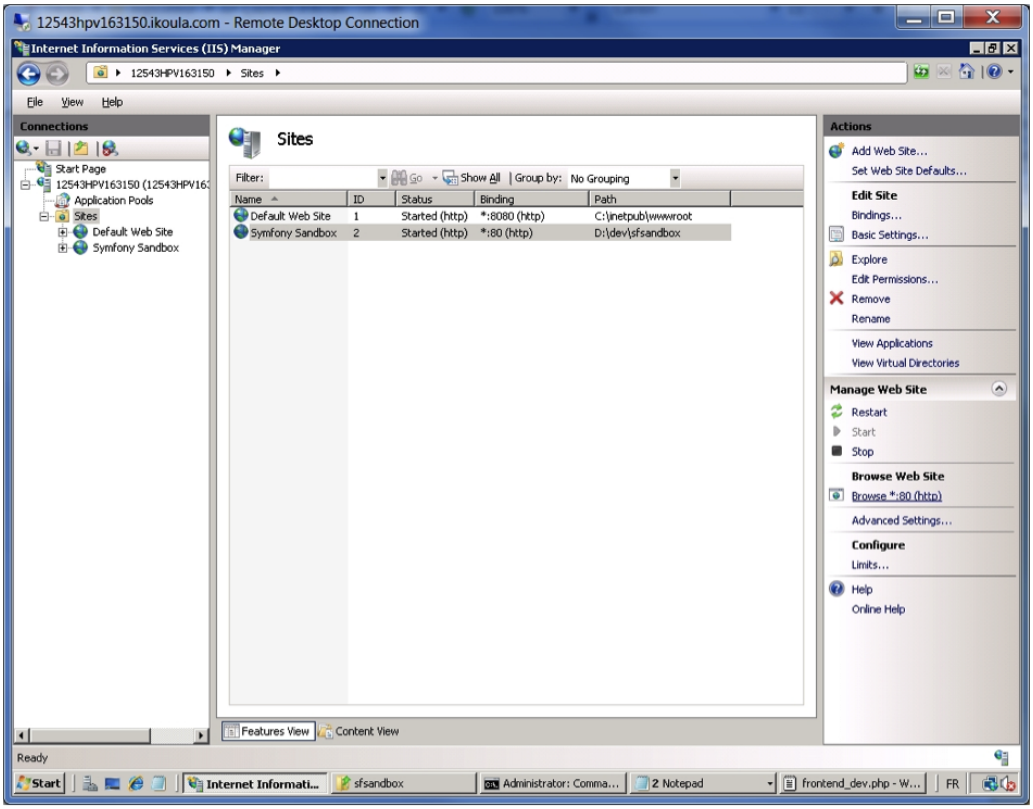
Open IIS Manager from Administration Tools. On the left pane, select the “Sites” icon, and right-click. Select Add Web Site from the popup menu. Enter, for instance, “Symfony Sandbox” as the Site name, `D:\dev\sfsandbox` for the Physical Path, and leave the other fields unchanged. You should see this dialog box:




Click OK. If a small x appears on the web site icon (in Features View / Sites), don’t hesitate to click “Restart” on the right pane to make it disappear.

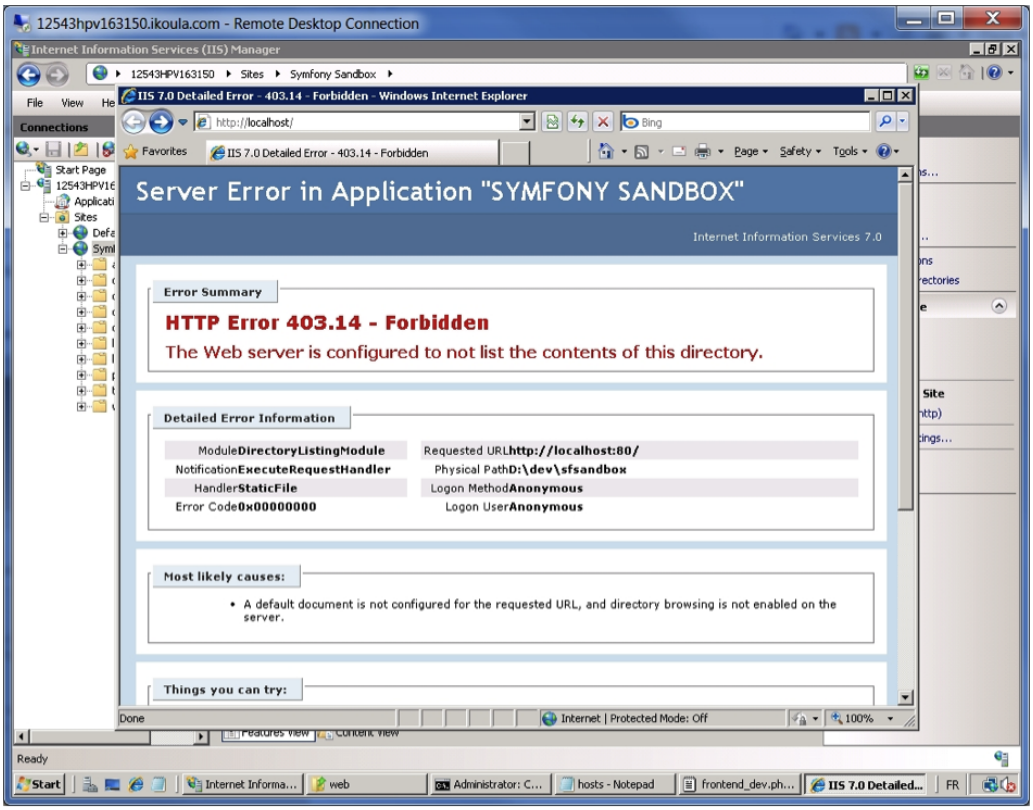
## Check if the Web Site is Answering

From IIS Manager, select the “Symfony Sandbox” site, and, on the Right pane, click on “Browse *.80 (http)”.

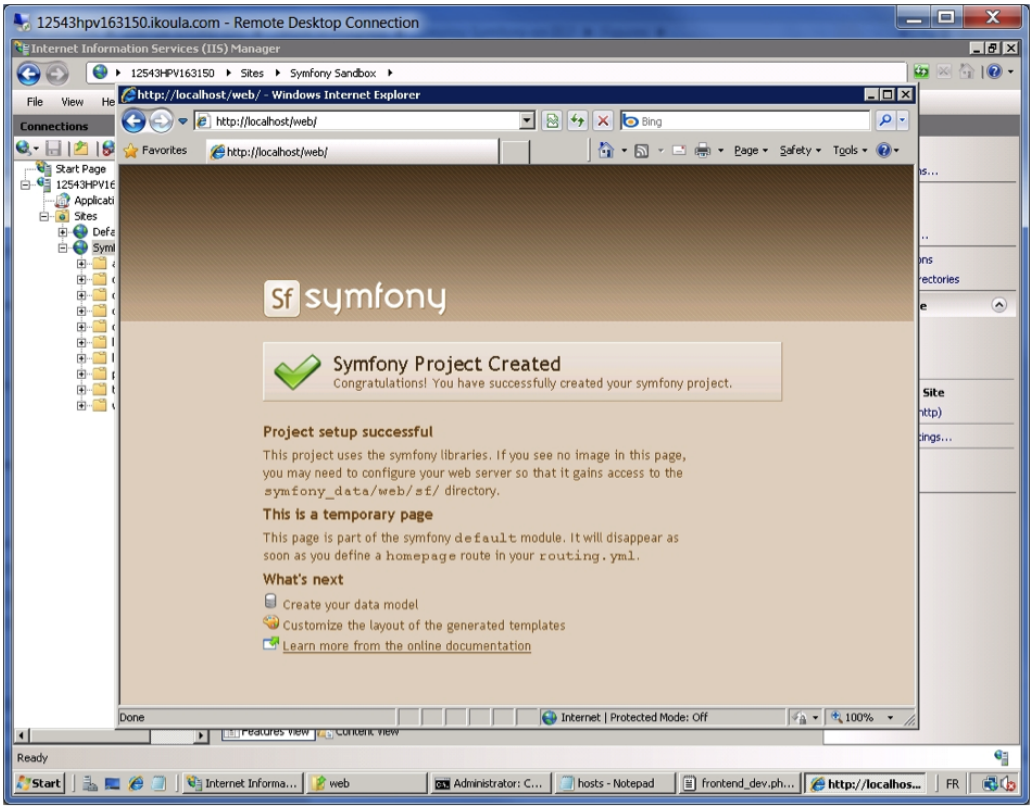



You should get an explicit error message, this is not unexpected: `HTTP Error 403.14 - Forbidden`. The Web server is configured to not list the contents of this directory.

This originates from the default web server configuration, which specifies that the contents of this directory should not be listed. Since no default file such as `index.php` or `index.html` exists at `D:\dev\sfsandbox`, the server correctly returns the "Forbidden" error message. Don't be afraid.



Type `http://localhost/web` in the URL bar of your browser, instead of just `http://localhost`. You now should see your browser, by default Internet Explorer, displaying “Symfony Project Created”:




By the way, you may see a light yellow band at the top saying “Intranet settings are now turned off by default”. Intranet settings are less secure than Internet settings. Click for options. Don’t be afraid of this message.

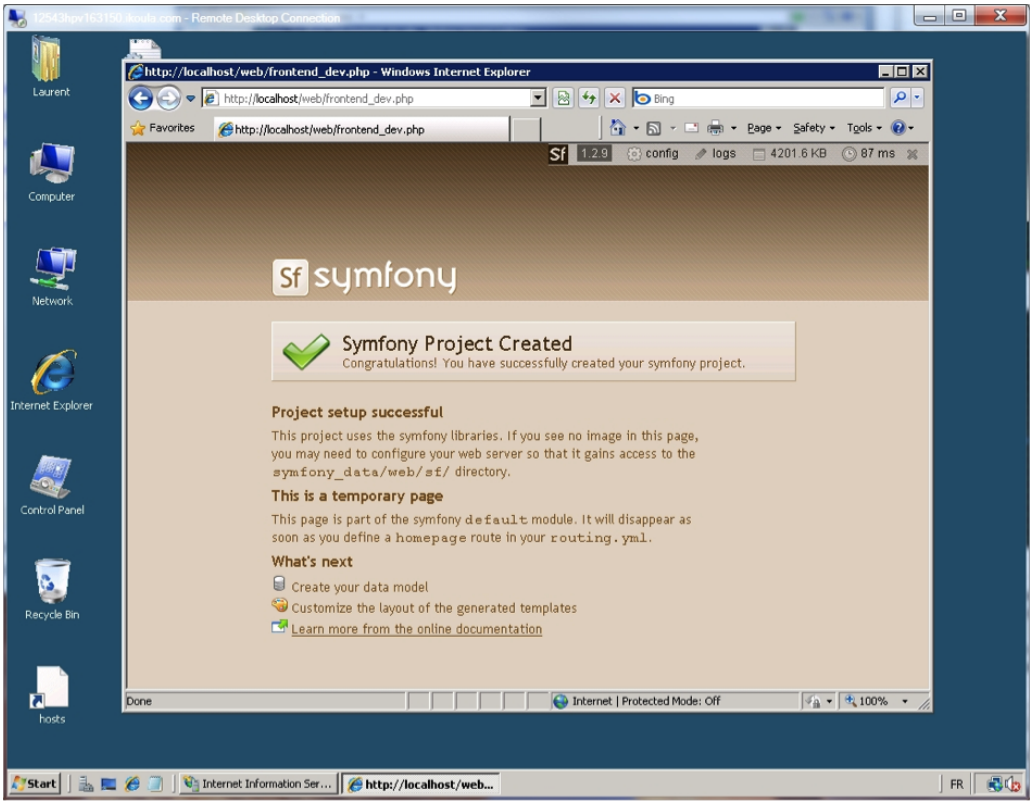
To close it permanently, just right-click the yellow band, and select the appropriate option.

This screen confirms that the default `index.php` page was correctly loaded from `D:\dev\sfsandbox\web\index.php`, correctly executed, and that symfony libraries are correctly configured.

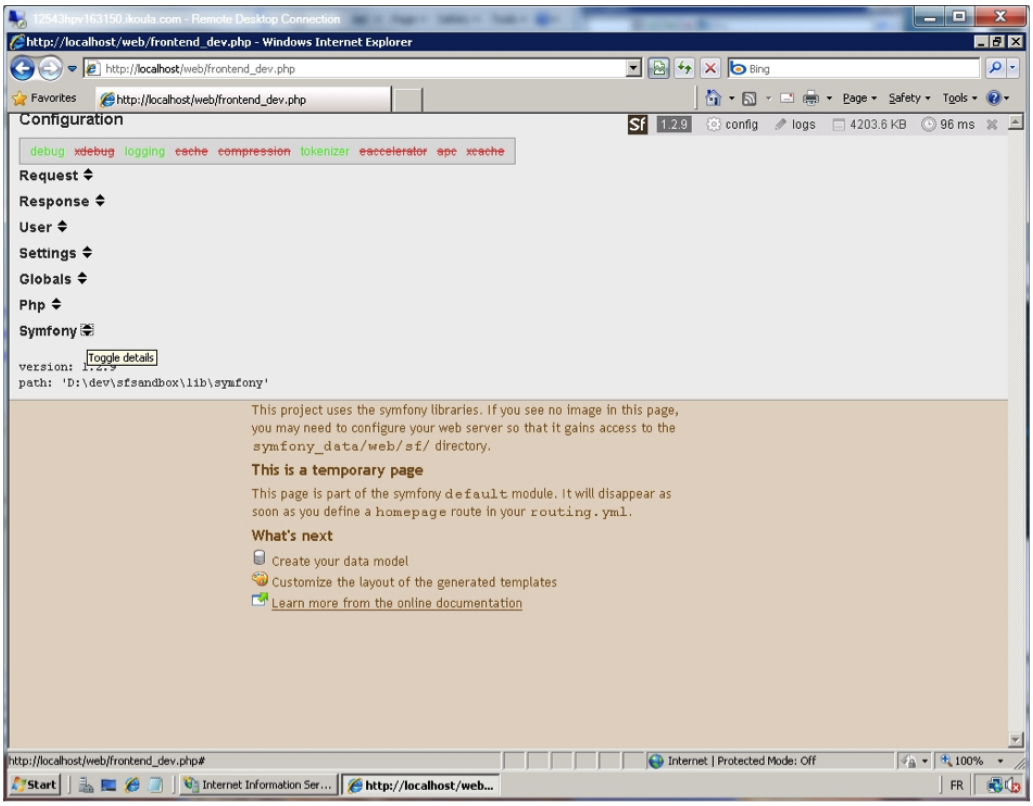
We need to perform one last task before beginning to play with the symfony sandbox: configure the front-end web page by importing URL rewrite rules. These rules are implemented as `.htaccess` files and can be controlled in just a few clicks in the IIS Manager.

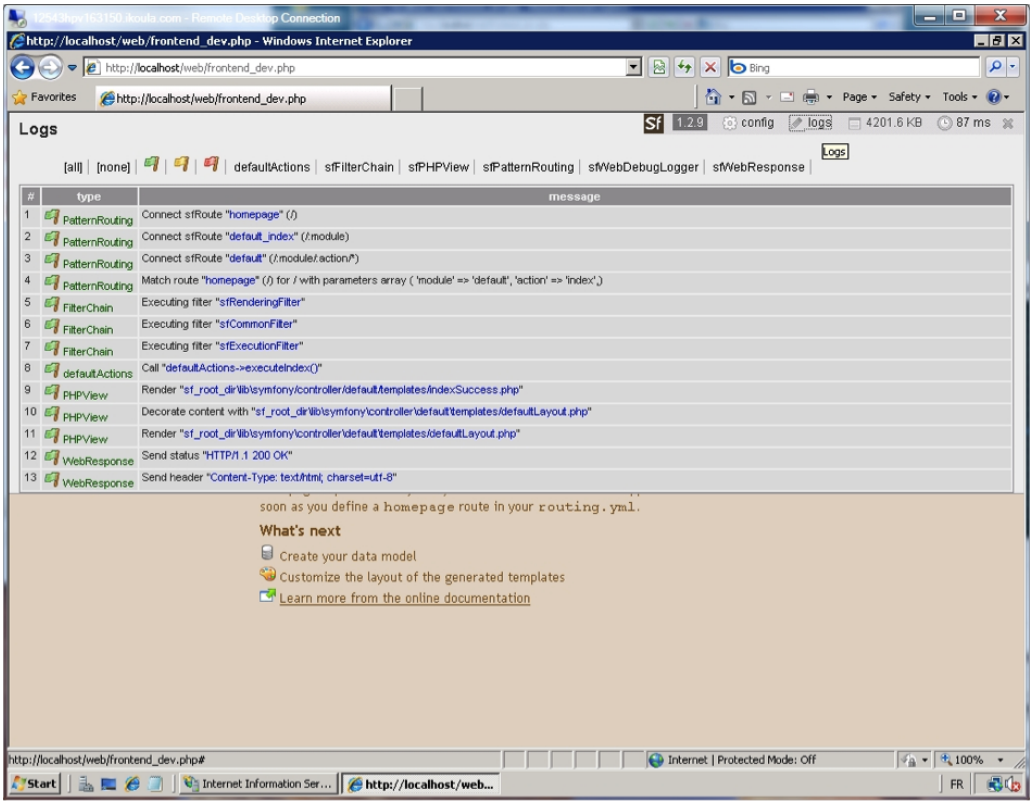
## Sandbox: Web Front-end Configuration

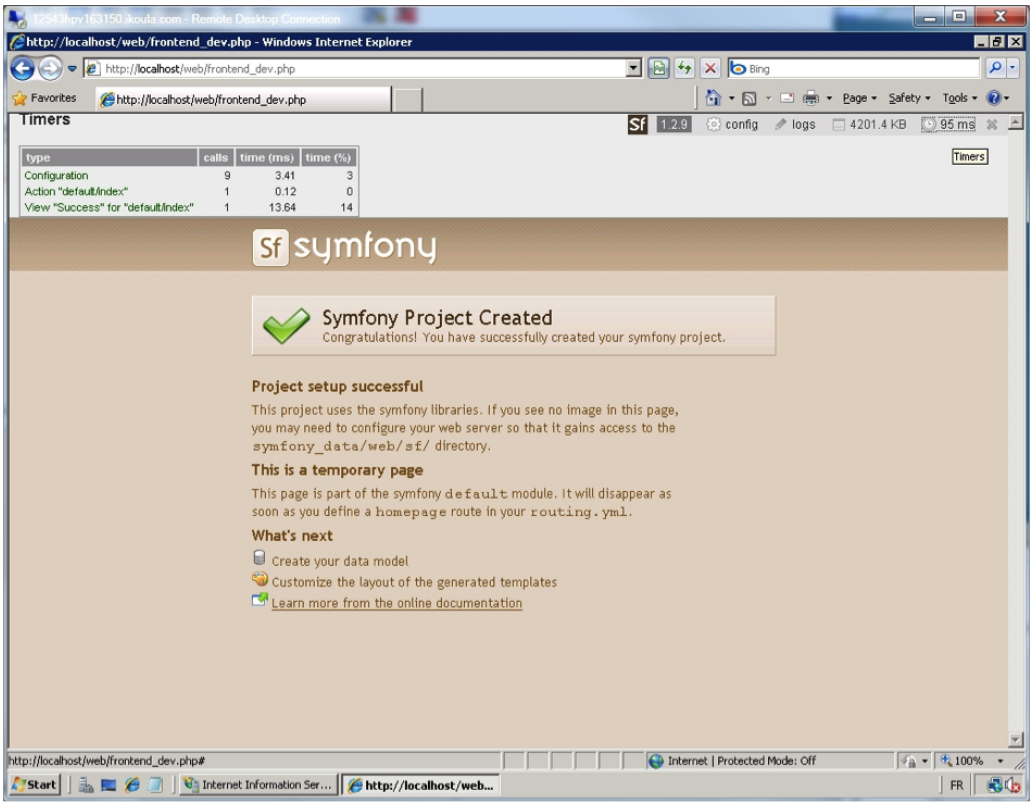
We want to configure the sandbox application front-end in order to begin to play with the real symfony stuff. By default, the front-end page can be reached and executes correctly when requested from the local machine (i.e. the `localhost` name or the `127.0.0.1` address).




Let's explore the "configuration", "logs" and "timers" web debug panels to ensure that the sandbox is fully functional on Windows Server 2008.






While we could try to request the sandbox application from the Internet or from a remote IP address, the sandbox is mostly designed as a tool to learn the symfony framework on the local machine. Hence, we'll cover details related to remote access in the last section: Project: web Front-end Configuration.

# Creation of a new symfony Project

Creating a symfony project environment for real development purposes is almost as straightforward as the installation of the sandbox. We'll see the whole installation process in a simplified procedure, as it is equivalent to the sandbox's installation and deployment.

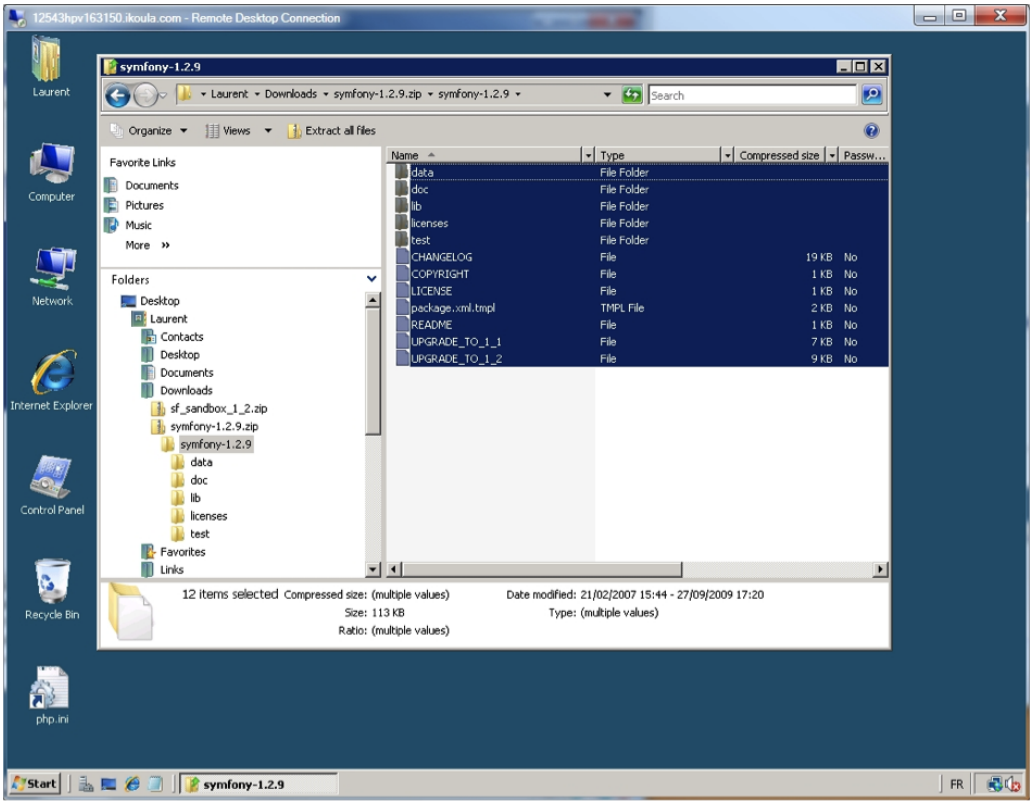
The difference is that, in this "project" section, we'll focus on the configuration of the web application to make it work from anywhere on the Internet.

Like the sandbox, the symfony project comes pre-configured to use SQLite as a database engine. This was installed and configured earlier in this chapter.

## Download, create a Directory and copy the Files

Each version of symfony can be downloaded as a .zip file and then used to create a project from scratch.

Download the archive containing the library from the symfony website[134]. Next, extract the contained directory to a temporary location, such as the "downloads" directory.




Now we need to create a directory tree for the final destination of the project. This is a bit more complicated than the sandbox.

## Directory Tree Setup

Let's create a directory tree for the project. Start from the volume root, `D:` for instance.

Create a `\dev` directory on `D:`, and create another directory called `sfproject` there:

*Listing 11-10*

```
D:
MD dev
CD dev
```

134. `http://www.symfony-project.org/get/symfony-1.3.0.zip`

```
MD sfproject
CD sfproject
```

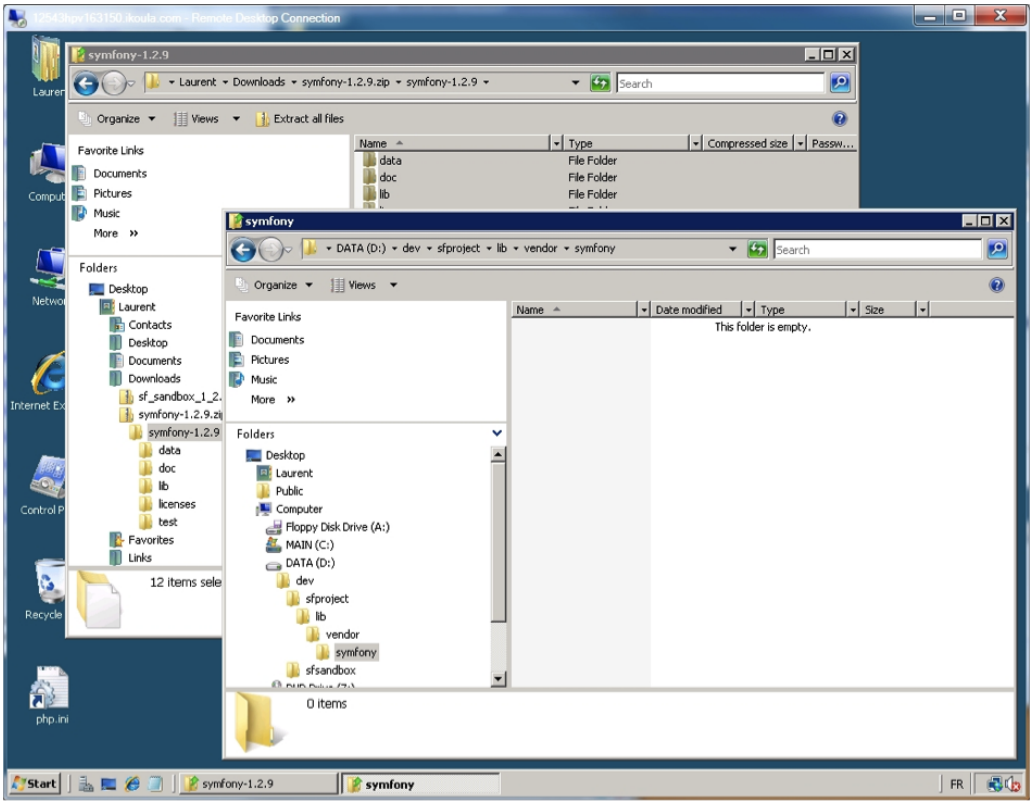
We are now in: `D:\dev\sfproject`

From there, create a subdirectory tree, by creating the `lib`, `vendor` and `symfony` directories in a cascading manner:

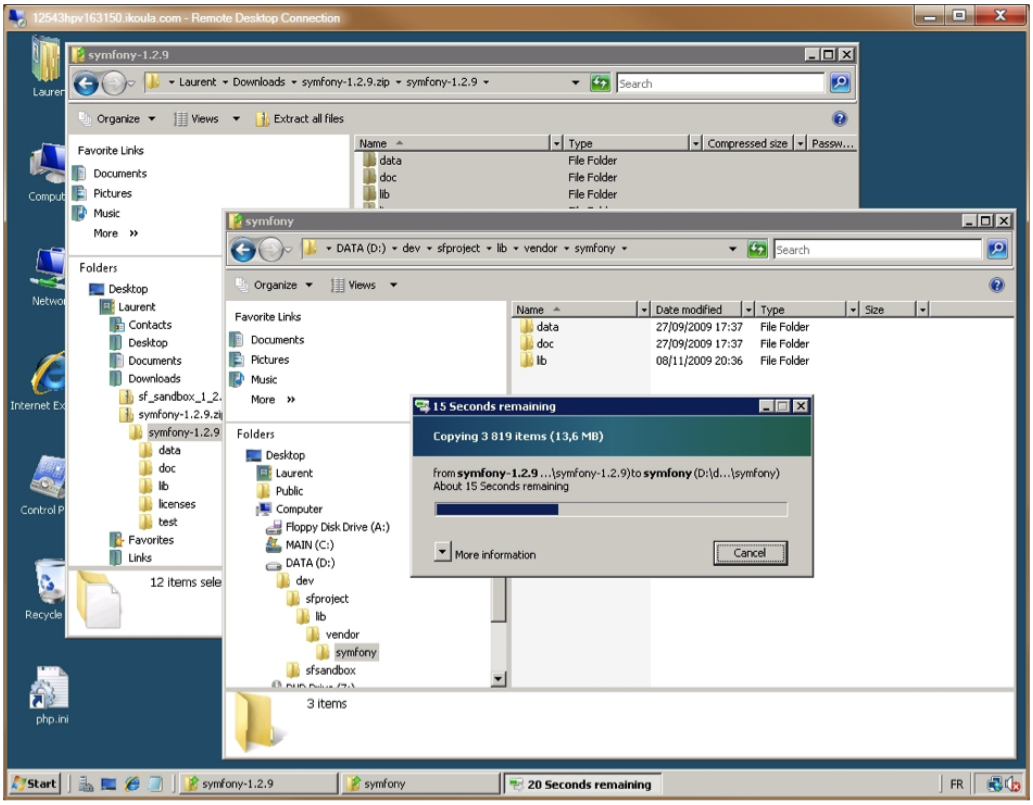
*Listing 11-11*

```
MD lib
CD lib
MD vendor
CD vendor
MD symfony
CD symfony
```

We now are in: `D:\dev\sfproject\lib\vendor\symfony`




Select all files (`CTRL-A` in Windows Explorer) from your download location (source), and copy from Downloads to `D:\dev\sfproject\lib\vendor\symfony`. You should see 3819 items copied to the destination directory:




## Creation and Initialization

Open the command prompt. Change to the `D:\dev\sfproject` directory and execute the following command:

```
PHP lib\vendor\symfony\data\bin\symfony -V
```

*Listing 11-12*

This should return:

```
symfony version 1.3.0 (D:\dev\sfproject\lib\vendor\symfony\lib)
```

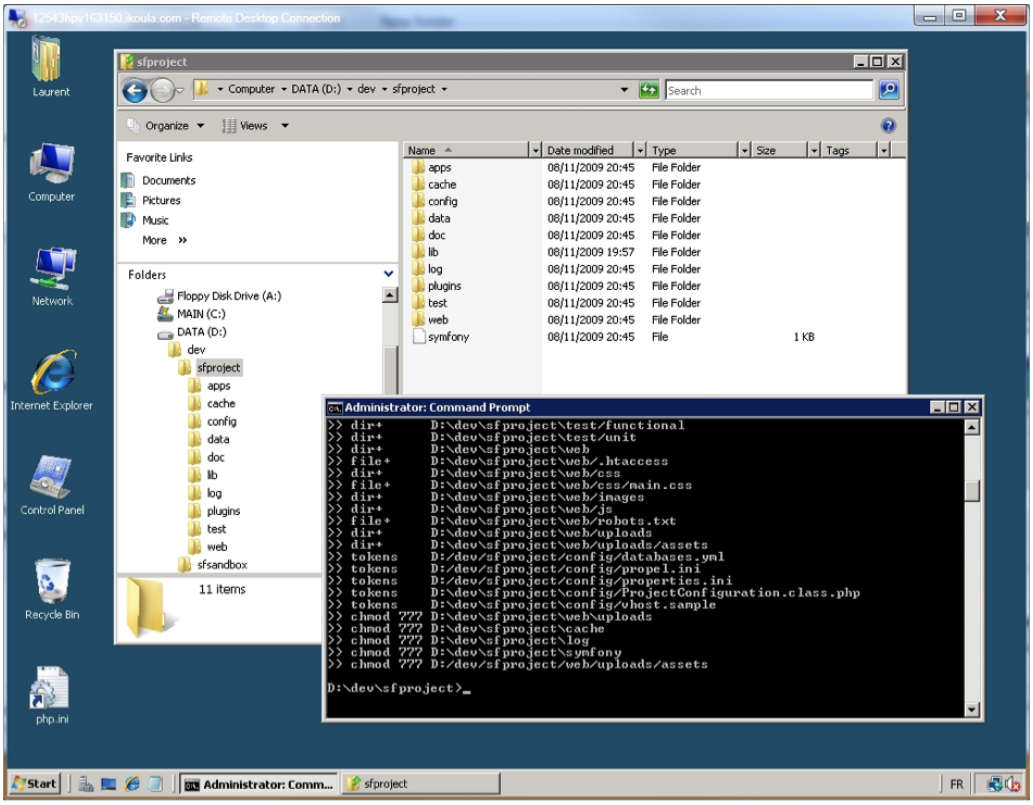
*Listing 11-13*

To initialize the project, just run the following PHP command line:

```
PHP lib\vendor\symfony\data\bin\symfony generate:project sfproject
```

*Listing 11-14*

This should return a list of file operations, including some `chmod 777` commands:




Still within the command prompt, create a symfony application by running the following command:

*Listing 11-15*
```
PHP lib\vendor\symfony\data\bin\symfony generate:app sfapp
```

Again, this should return a list of file operations, including some `chmod 777` commands.

From this point, instead of typing `PHP lib\vendor\symfony\data\bin\symfony` each time it's needed, copy the `symfony.bat` file from its origin:

*Listing 11-16*
```
copy lib\vendor\symfony\data\bin\symfony.bat
```

We now have a convenient command to run at the command-line prompt in `D:\dev\sfproject`.

Still in `D:\dev\sfproject`, we can run the now-classic command:

*Listing 11-17*
```
symfony -V
```

to get the classic answer:

*Listing 11-18*
```
symfony version 1.3.0 (D:\dev\sfproject\lib\vendor\symfony\lib)
```

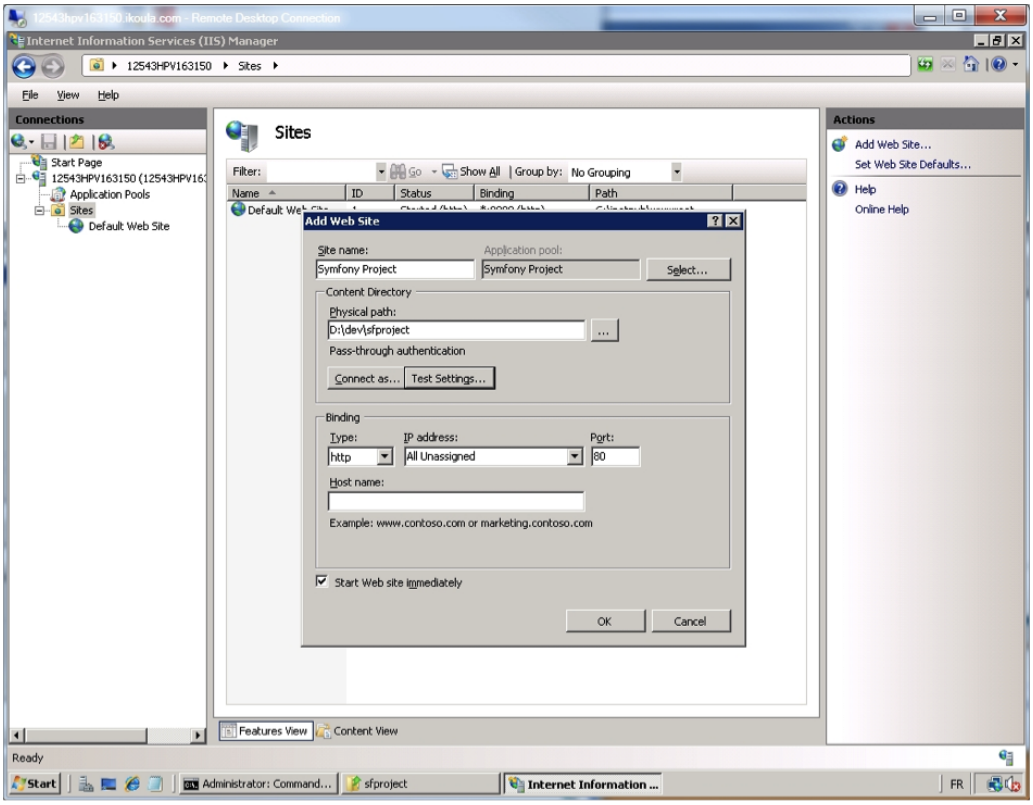
## Web Application Creation

In the following lines, we assume that you've read the "Sandbox: Web Application Creation" preliminary steps to reconfigure the "Default Web Site", so that it does not interfere on port 80.

### Add a new Web Site for the Project

Open IIS Manager from Administration Tools. On the left pane, select the "Sites" icon, and right-click. Select "Add Web Site" from the popup menu. Enter, for instance, "Symfony

Project" as the Site name, `D:\dev\sfproject` for the "Physical Path", and leave other fields unchanged; you should see this dialog box:




Click OK. If a small `x` appears on the web site icon (in Features View / Sites), don't hesitate to click "Restart" on the right pane to make it disappear.

## Check if the web Site is Answering

From IIS Manager, select the "Symfony Project" site, and, on the Right pane, click "Browse *.80 (http)".

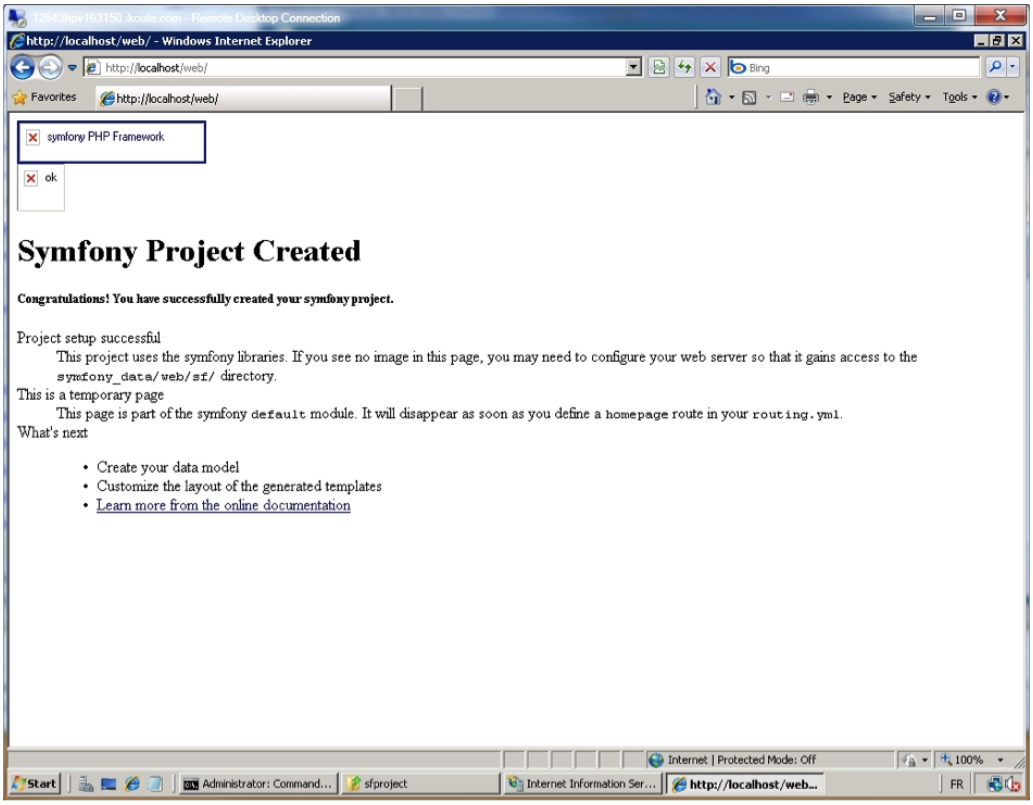
You should get the same explicit error message as you had when trying the sandbox:

```
HTTP Error 403.14 - Forbidden
```

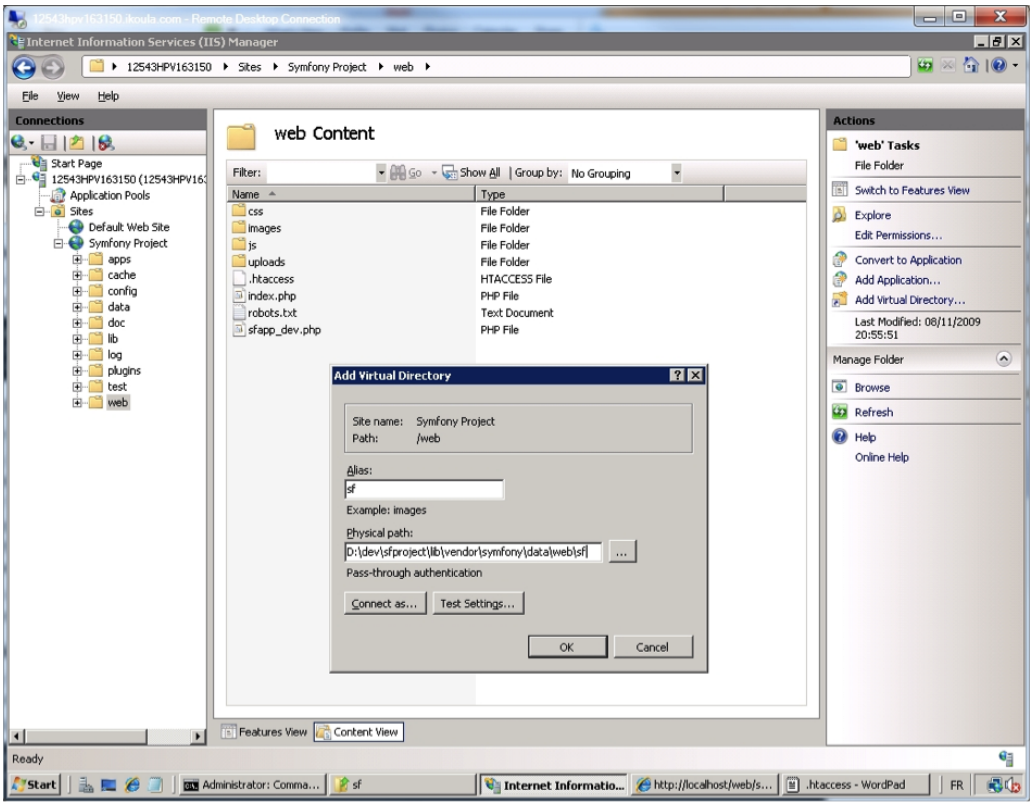
*Listing 11-19*

The Web server is configured to not list the contents of this directory.

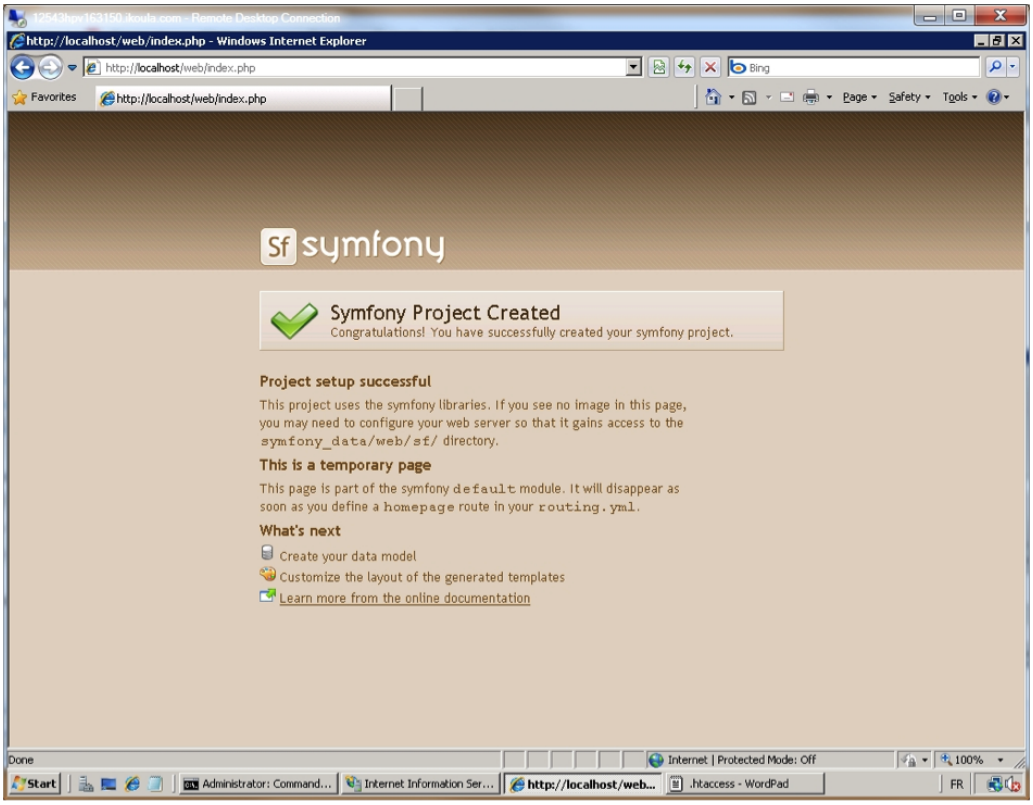
Type `http://localhost/web` in the URL bar of your browser You should now see the "Symfony Project Created" page, but with one slight difference from the same page resulting from sandbox initialization: there are no images:



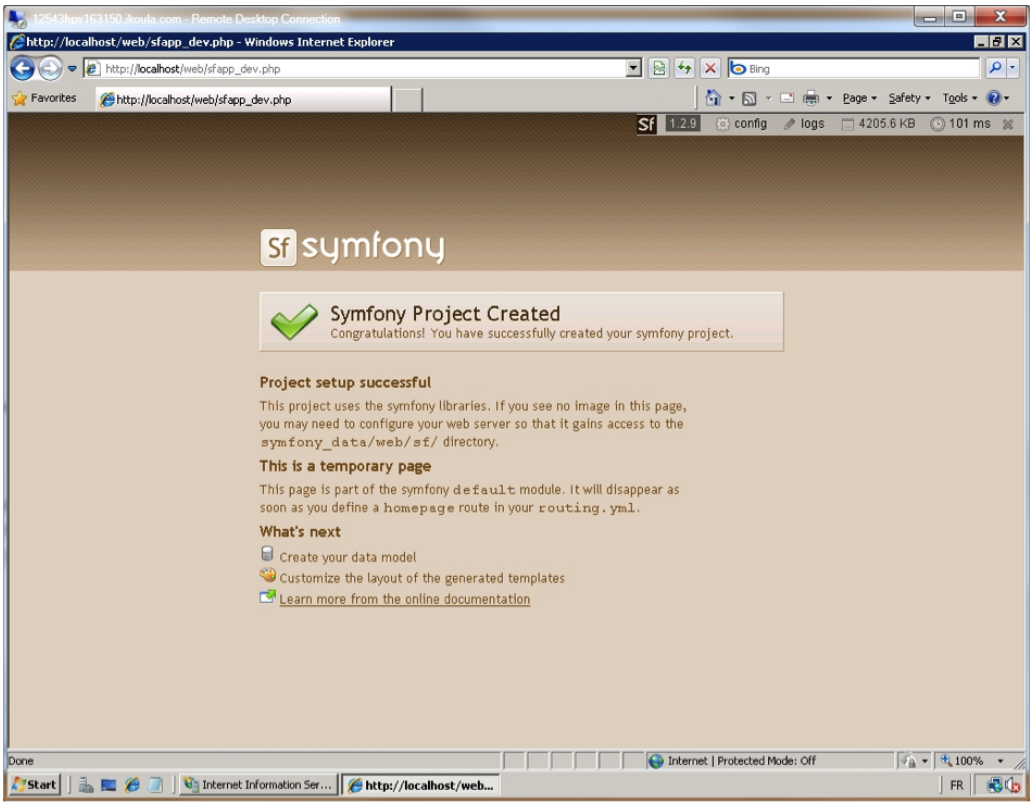

The images are not here for the moment, though they're located in an `sf` directory in the symfony library. It's easy to link them to the `/web` directory by adding a virtual directory in `/web`, named `sf`, and pointing it to `D:\dev\sfproject\lib\vendor\symfony\data\web\sf`.



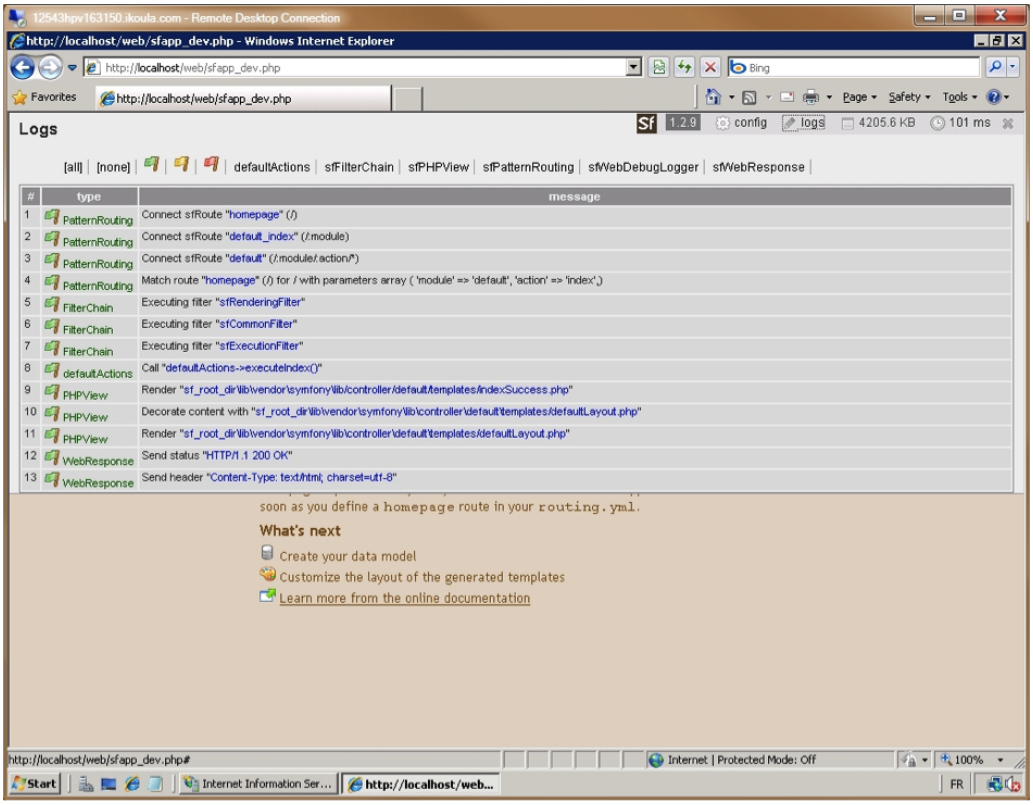

Now we have the regular "Symfony Project Created" page with images as expected:




And finally, the whole symfony application is working. From the web browser, enter the URL of the web application, i.e. `http://localhost/web/sfapp_dev.php`:




Let's perform one last test while in local mode: check the "configuration", "logs" and "timers" web debug panels to ensure that the project is fully functional.

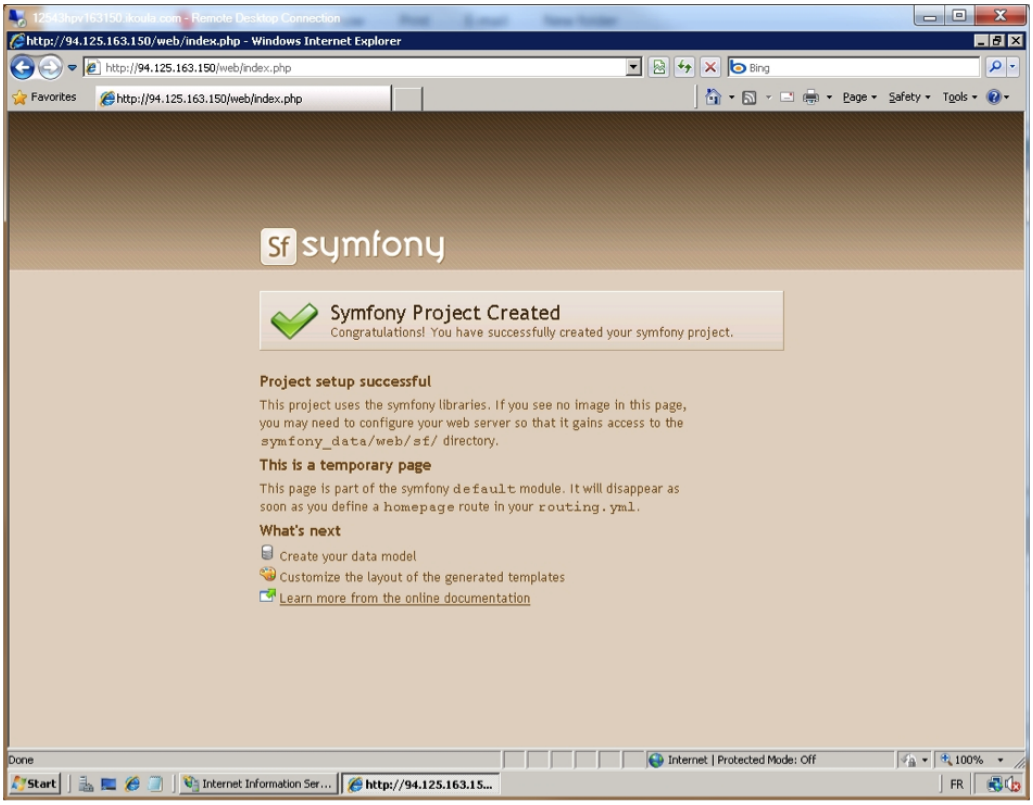


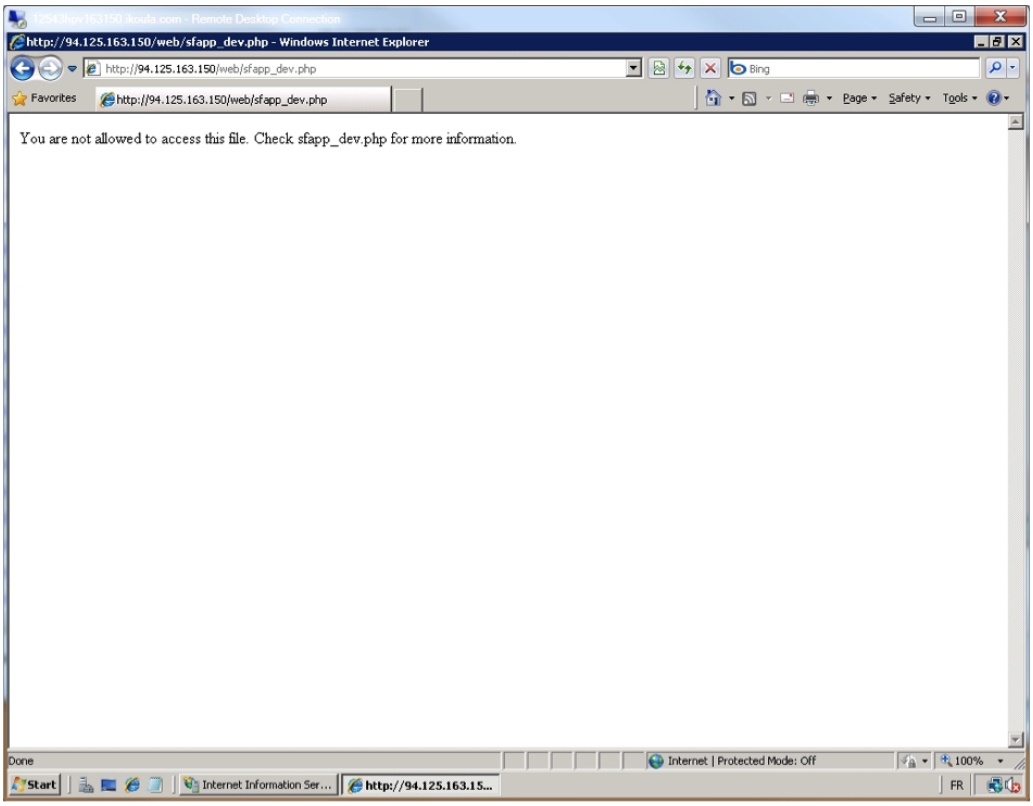

## Application Configuration for Internet-ready Applications

Our generic symfony project is now working locally, like the sandbox, from the local host server, located at `http://localhost` or `http://127.0.0.1`.

Now, we'd like to be able to access the application from the Internet.

The default configuration of the project protects the application from being executed from a remote location, though, in reality it should be ok to access both the `index.php` and `sfapp_dev.php` files. Let's execute the project from the web browser, using the server's external IP address (e.g. `94.125.163.150`) and the FQDN of our Virtual Dedicated Server (e.g. `12543hpv163150.ikoula.com`). You can even use both addresses from inside the server, as they're not mapped to `127.0.0.1`:




As we said before, access to `index.php` and `sfapp_dev.php` from a remote location is ok. The execution of `sfapp_dev.php`, however fails, as it is not allowed by default. This prevents potentially malicious users from accessing your development environment, which contains potentially sensitive information about your project. You can edit the `sfapp_dev.php` file to make it work, but this is strongly discouraged.

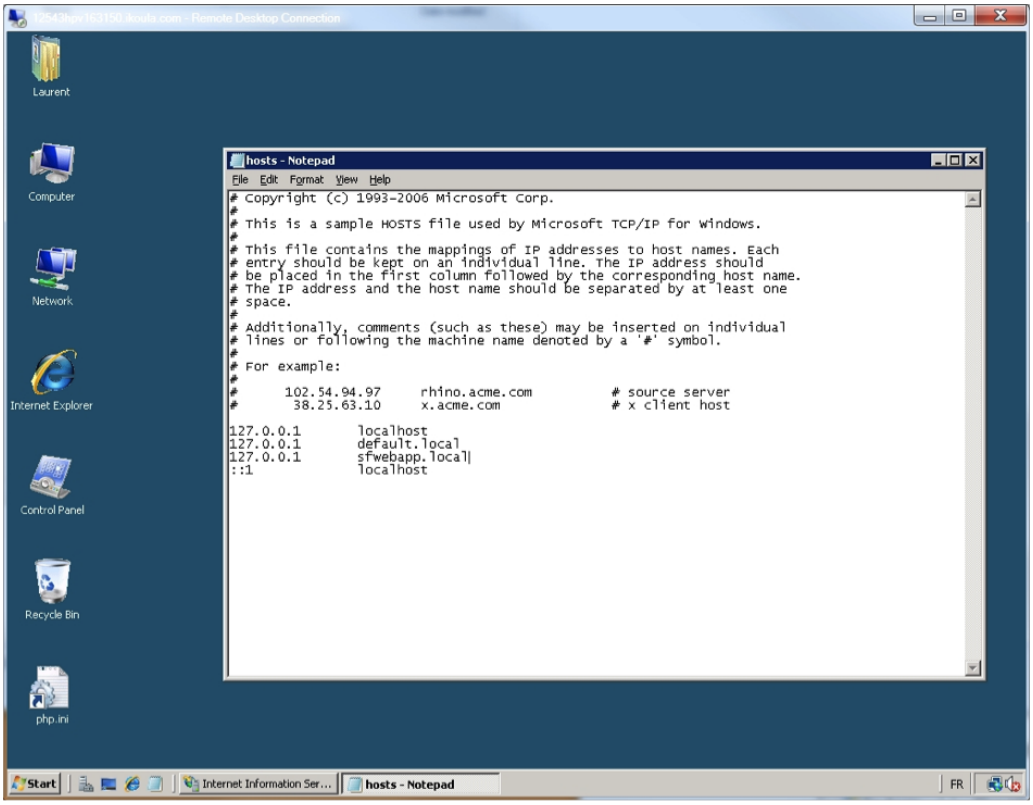
Finally, we can simulate a real domain by editing the “hosts” file.

This file performs the local FQDN name resolution without needing to install the DNS service on Windows. The DNS service is available on all editions of Windows Server 2008 R2, and also in Windows Server 2008 Standard, Enterprise and Datacenter editions.

On Windows x64 operating systems, the "hosts" file is located by default in: `C:\Windows\SysWOW64\Drivers\etc`

The "hosts" file is pre-populated to have the machine resolve `localhost` to `127.0.0.1` in IPv4, and `::1` in IPv6.

Let's add a fake real domain name, like `sfwebapp.local`, and have it resolve locally.




Your symfony project now runs on the Web, without DNS, from a web browser session executed from within the Web server.

Chapter 12

# Developing for Facebook

*by Fabrice Bernhard*

Facebook, with more than 350 million members, has become the standard in social websites on the Internet. One of its most interesting features is the “Facebook Platform”, an API which enables developers to create applications inside the Facebook website as well as connect other websites with the Facebook authentication system and social graph.

Since the Facebook frontend is in PHP, it is no wonder that the official client library of this API is a PHP library. This de facto makes symfony a logical solution for developing quick and clean Facebook applications or Facebook Connect sites. But more than that, developing for Facebook really shows how you can leverage symfony’s functionalities to gain precious time while keeping high standards of quality. This is what will be covered here in depth: after a brief summary of what the Facebook API is and how it can be used, we will cover how to use symfony at its best when developing Facebook applications, how to benefit from the community’s efforts and the `sfFacebookConnectPlugin`, demonstrate it through a simple “Hello you!” application and finally give tips and tricks to solve the most common problems.

## Developing for Facebook

Although the API is basically the same in both cases, there are two very different use-cases: creating a Facebook application inside Facebook and implementing Facebook Connect on an external website.

### Facebook Applications

Facebook applications are web applications inside Facebook. Their main quality is to be directly embedded into the 300 million users strong social website, therefore allowing any viral application grow at incredible speed. Farmville is the biggest and latest example, with more than 60 million monthly active users and 2 million fans gained in a few months! This is the equivalent of the French population coming back every month to work on their virtual farm! Facebook applications interact with the Facebook website and its social graph in different ways. Here is a brief look a the different places where the Facebook application will be able to appear:

#### The Canvas

The canvas will usually be the main part of the application. It is basically a small website embedded inside the Facebook frame.

### The Profile Tab

The application can also be inside a tab on a user's profile or a fan page. The main limitations are:

- only one page. It is not possible to define links to sub-pages in the tab.
- no dynamic flash or JavaScript at loading time. To provide dynamic functionalities, the application has to wait for the user to interact on the page by clicking on a link or button.

### The Profile Box

This is more of a remain of the old Facebook, which nobody really uses anymore. It is used to display some information in a box that can be found in the "Boxes" tab of the profile.

### The Information Tab's Addendum

Some static information linked to a specific user and the application can be displayed in the information tab of the user's profile. This appears just under the user's age, address, and curriculum.

### Publishing Notices and information in the News Stream

The application can publish news, links, pictures, videos in the news stream, on a user's friend's wall, or directly modify the user's status.

### The Information page

This is the "profile page" of the application, automatically created by Facebook. This is where the creator of the application will be able to interact with their users in the usual Facebook way. This generally concerns the marketing team more directly than the development team.

## Facebook Connect

Facebook Connect enables any website to bring some of the great functionalities of Facebook to its own users. Already "connected" websites can be recognized by the presence of a big blue "Connect with Facebook" button. The most famous include digg.com, cnet.com, netvibes.com, yelp.com, etc. Here is the list of the four main reasons to bring Facebook connect to an existing site.

### One-click Authentication System

Just like OpenID, Facebook Connect gives websites the opportunity to provide automatic-login using their Facebook session. Once the "connection" between the website and Facebook has been approved by the user, the Facebook session is automatically provided to the website, saving him the cost of yet another registration to do and password to remember.

### Get more Information about the User

One other key feature of Facebook Connect is the quantity of information provided. While a user will generally upload a minimum set of information to a new website, Facebook Connect gives the opportunity to quickly get interesting additional information like name, age, sex, location, profile picture, etc. enriching the website. The terms of use of Facebook Connect clearly remind that one should not store any personal information about the user without the user explicitly agreeing about it, but the information provided can be used to fill forms and ask for confirmation in a click. Additionally, the website can rely on public information like name and profile picture without needing to store them.

### Viral Communication using the News Feed

The ability to interact with the user's new feed, invite friends or publish on friend's walls lets the website use the full viral potential of Facebook to communicate. Any website with some social component can really benefit from this feature, as long as the information published in the Facebook feed has some social value that might interest friends and friends of friends.

### Take advantage of the existing Social Graph

For a website whose service relies on a social graph (like a network of friends or acquaintances), the cost to build a first community, with enough links between users to interact and benefit from the service, is really high. By giving easy access to the list of friends of a user, Facebook Connect dramatically reduces this cost, removing the need to search for "already registered friends".

## Setting up a first Project using `sfFacebookConnectPlugin`

### Create the Facebook Application

To begin, a Facebook account is needed with the "Developer"[135] application installed. To create the application, the only information needed is a name. Once this is done, no further configuration is needed.

### Install and Configure `sfFacebookConnectPlugin`

The next step is to link Facebook's users with `sfGuard` users. This is the main feature of the `sfFacebookConnectPlugin`, which I started and to which other symfony developers have quickly contributed. Once the plugin is installed, there is an easy but necessary configuration step. The API key, application secret, and application ID need to be setup in the `app.yml` file:

*Listing 12-1*

```
# default values
all:
  facebook:
    api_key: xxx
    api_secret: xxx
    api_id: xxx
    redirect_after_connect: false
    redirect_after_connect_url: ''
    connect_signin_url: 'sfFacebookConnectAuth/signin'
    app_url: '/my-app'
    guard_adapter: ~
    js_framework: none # none, jQuery or prototype.

  sf_guard_plugin:
    profile_class: sfGuardUserProfile
    profile_field_name: user_id
    profile_facebook_uid_name: facebook_uid # WARNING this column must be
of type varchar! 100000398093902 is a valid uid for example!
    profile_email_name: email
    profile_email_hash_name: email_hash
```

135. `http://www.facebook.com/developers`

```
  facebook_connect:
    load_routing:     true
    user_permissions: []
```

With older versions of symfony, remember to set the “load_routing” option to false, since it uses the new routing system.

## Configure a Facebook Application

If the project is a Facebook application, the only other important parameter will be the `app_url` which points to the relative path of the application on Facebook. For example, for the application `http://apps.facebook.com/my-app` the value of the `app_url` parameter will be `/my-app`.

## Configure a Facebook Connect Website

If the project is a Facebook Connect website, the other configuration parameters can be left with the default values most of the time:

- `redirect_after_connect` enables tweaking the behaviour after clicking on the “Connect with Facebook” button. By default the plugin reproduces the behaviour of `sfGuardPlugin` after registration.
- `js_framework` can be used to specify a specific JS framework to use. It is highly recommended to use a JS framework such as jQuery on Facebook Connect sites since the JavaScript of Facebook Connect is quite heavy and can cause fatal errors (!) on IE6 if not loaded at the right moment.
- `user_permissions` is the array of permissions that will be given to new Facebook Connect users.

## Connecting sfGuard with Facebook

The link between a Facebook user and the `sfGuardPlugin` system is done quite logically using a `facebook_uid` column in the `Profile` table. The plugin assumes that the link between the `sfGuardUser` and its profile is done using the `getProfile()` method. This is the default behaviour with `sfPropelGuardPlugin` but needs to be configured as such with `sfDoctrineGuardPlugin`. Here are possible `schema.yml`:

For Propel:

*Listing 12-2*

```
sf_guard_user_profile:
  _attributes: { phpName: UserProfile }
  id:
  user_id:            { type: integer, foreignTable: sf_guard_user,
foreignReference: id, onDelete: cascade }
  first_name:         { type: varchar, size: 30 }
  last_name:          { type: varchar, size: 30 }
  facebook_uid:       { type: varchar, size: 20 }
  email:              { type: varchar, size: 255 }
  email_hash:         { type: varchar, size: 255 }
  _uniques:
    facebook_uid_index: [facebook_uid]
    email_index:        [email]
    email_hash_index:   [email_hash]
```

For Doctrine:

Listing 12-3

```
sfGuardUserProfile:
  tableName:     sf_guard_user_profile
  columns:
    user_id:          { type: integer(4), notnull: true }
    first_name:       { type: string(30) }
    last_name:        { type: string(30) }
    facebook_uid:     { type: string(20) }
    email:            { type: string(255) }
    email_hash:       { type: string(255) }
  indexes:
    facebook_uid_index:
      fields: [facebook_uid]
      unique: true
    email_index:
      fields: [email]
      unique: true
    email_hash_index:
      fields: [email_hash]
      unique: true
  relations:
    sfGuardUser:
      type: one
      foreignType: one
      class: sfGuardUser
      local: user_id
      foreign: id
      onDelete: cascade
      foreignAlias: Profile
```

What if the project uses Doctrine and the `foreignAlias` is not `Profile`. In that case the plugin will not work. But a simple `getProfile()` method in the `sfGuardUser.class.php` which points to the `Profile` table will solve the problem!

Please note that the `facebook_uid` column should be `varchar`, because new profiles on Facebook have `uids` above `10^15`. Better play safe with an indexed `varchar` column than try to make `bigints` work with different ORMs.

The other two columns are less important: `email` and `email_hash` are only required in the case of a Facebook Connect website with existing users. In that case Facebook provides a complicated process to try to associate existing accounts with new Facebook connect accounts using a hash of the email. Of course the process is made simple thanks to a task provided by the `sfFacebookConnectPlugin`, which is described in the last part of this chapter.

## Choosing between FBML and XFBML: Problem solved by symfony

Now that everything is setup, we can start to code the actual application. Facebook offers many special tags that can render entire functionalities, like an “invite friends” form or a fully-working comment system. These tags are called FBML or XFBML tags. FBML and XFBML tags are quite similar but the choice depends on whether the application is rendered inside Facebook or not. If the project is a Facebook connect website, there is only one choice: XFBML. If it is a Facebook application, there are two choices:

- Embed the application as a real IFrame inside the Facebook application page and use XFBML inside this IFrame;
- Let Facebook embed it transparently inside the page, and use FBML.

Facebook encourages developers to use their “transparent embedding” or the so-called “FBML application”. Indeed , it has some interesting features:

- No Iframe, which is always complicated to manage since you need to keep it in mind if your links concern the Iframe or the parent window;
- Special tags called FBML tags are interpreted automatically by the Facebook server and enable you to display private information concerning the user without having to communicate with the Facebook server beforehand;
- No need to pass the Facebook session from page to page manually.

But FBML has some serious drawbacks too:

- Every JavaScript is embedded inside a sandbox, making it impossible to use outside libraries like a Google Maps, jQuery or any statistics system other than Google Analytics officially supported by Facebook;
- It claims to be faster since some API calls can be replaced by FBML tags. However if the application is light, hosting it on its own server will be much faster;
- It is harder to debug, especially error 500 which are caught by Facebook and replaced by a standard error.

So what is the recommended choice? The good news is that, with symfony and the `sfFacebookConnectPlugin`, there is no choice to make! It is possible to write agnostic applications and switch indifferently from an IFrame to an embedded application to a Facebook Connect website with the same code. This is possible because, technically, the main difference actually is in the layout... which is very easy to switch in symfony. Here are the examples of the two different layouts:

The layout for an FBML application:

*Listing 12-4*

```
<?php sfConfig::set('sf_web_debug', false); ?>
<fb:title><?php echo sfContext::getInstance()->getResponse()->getTitle()
?></fb:title>
<?php echo $sf_content ?>
```

The layout for an XFBML application or Facebook Connect website:

*Listing 12-5*

```
<?php use_helper('sfFacebookConnect')?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:fb="http://www.facebook.com/2008/fbml">
  <head>
    <?php include_http_metas() ?>
    <?php include_metas() ?>
    <?php include_title() ?>
    <script type="text/javascript" src="http://www.symfony-project.org/
sfFacebookConnectPlugin/js/animation/animation.js"></script>
  </head>
  <body>
    <?php echo $sf_content ?>
    <?php echo include_facebook_connect_script() ?>
  </body>
</html>
```

To switch automatically between both, simply add this to you `actions.class.php` file:

*Listing 12-6*

```
public function preExecute()
{
  if (sfFacebook::isInsideFacebook())
  {
    $this->setLayout('layout_fbml');
  }
  else
  {
    $this->setLayout('layout_connect');
  }
}
```

There is one small difference between FBML and XFBML which is not located in the layout: FBML tags can be closed, not XFBML ones. So just replace tags like:

```
<fb:profile-pic uid="12345" size="normal" width="400" />
```

*Listing 12-7*

by:

```
<fb:profile-pic uid="12345" size="normal" width="400"></fb:profile-pic>
```

*Listing 12-8*

Of course, to do this the application needs to be configured also as a Facebook Connect application in the developer's settings, even if the application is only intended for FBML purposes. But, the enormous advantage of doing this is the possibility to test the application locally. If you are creating a Facebook application and plan to use FBML tags, which is almost inevitable, the only way to view the result is to put the code online and see the result directly rendered in Facebook! Fortunately, thanks to Facebook Connect, XFBML tags can be rendered outside of facebook.com. And, as was just described, the only difference between FBML and XFBML tags is the layout. Therefore, this solution enables FBML tags to be rendered locally, as long as there is an Internet connection. Furthermore, with a development environment visible on the Internet, such as a server or a simple computer with port 80 open, even the parts relying on Facebook's authentication system will work outside of the facebook.com domain thanks to Facebook connect. This allows to test the entire application before uploading it on Facebook.

### The simple Hello You Application

With the following code in the home template, the "Hello You" application is finished:

```
<?php $sfGuardUser = sfFacebook::getSfGuardUserByFacebookSession(); ?>
Hello <fb:name uid="<?php echo
$sfGuardUser?$sfGuardUser->getProfile()->getFacebookUid():'' ?>"></fb:name>
```

*Listing 12-9*

The `sfFacebookConnectPlugin` automatically converts the visiting Facebook user into a `sfGuard` user. This enables a very easy integration with existing symfony code relying on the `sfGuardPlugin`.

## Facebook Connect

### How Facebook Connect works and different Integration Strategies

Facebook Connect basically shares its session with the website's session. This is done by copying authentication cookies from Facebook to the website by opening an IFrame on the

website that points to a Facebook page which itself opens an IFrame to the website. To do this, Facebook Connect needs to have access to the website, which makes it impossible to use or test Facebook Connect on a local server or in an Intranet. The entry point is the `xd_receiver.htm` file, which the `sfFacebookConnectPlugin` provides. Remember to use the `plugin:publish-assets` task to make this file accessible.

Once this is done, the Facebook official library is able to use the Facebook session. Additionally, `sfFacebookConnectPlugin` creates an `sfGuard` user linked to the Facebook session, which seamlessly integrates with the existing symfony website. This is why the plugin redirects by default to the `sfFacebookConnectAuth/signIn` action after the Facebook Connect button has been clicked and the Facebook Connect session has been validated. The plugin first looks for an existing user with the same Facebook UID or same Email hash (see "Connecting existing users with their Facebook account" at the end of the article). If none is found, a new user is created.

Another common strategy is not to create the user directly, but first redirect him to a specific registration form. There, one can use the Facebook session to pre-fill common information, for example, by adding the following code to the configuration method of the registration form:

*Listing 12-10*

```
public function setDefaultsFromFacebookSession()
{
  if ($fb_uid = sfFacebook::getAnyFacebookUid())
  {
    $ret = sfFacebook::getFacebookApi()->users_getInfo(
      array(
        $fb_uid
      ),
      array(
        'first_name',
        'last_name',
      )
    );

    if ($ret && count($ret)>0)
    {
      if (array_key_exists('first_name', $ret[0]))
      {
        $this->setDefault('first_name',$ret[0]['first_name']);
      }
      if (array_key_exists('last_name', $ret[0]))
      {
        $this->setDefault('last_name',$ret[0]['last_name']);
      }
    }
  }
```

To use the second strategy, simply specify in the `app.yml` file to redirect after Facebook Connect and the route to use for the redirection:

*Listing 12-11*

```
# default values
all:
  facebook:
    redirect_after_connect: true
    redirect_after_connect_url: '@register_with_facebook'
```

### The Facebook Connect Filter

Another important feature of Facebook Connect is that Facebook users are very often logged into Facebook when browsing the Internet. This is where the `sfFacebookConnectRememberMeFilter` proves very useful. If a user visits the website and is already logged into Facebook, the `sfFacebookConnectRememberMeFilter` will automatically log them into the website just as the "Remember me" filter does.

*Listing 12-12*

```
$sfGuardUser = sfFacebook::getSfGuardUserByFacebookSession();
if ($sfGuardUser)
{
  $this->getContext()->getUser()->signIn($sfGuardUser, true);
}
```

However this has one serious drawback: users can no longer logout from the website, since as long as they are connected on Facebook, they will be logged back automatically. Use this feature with caution.

### Clean implementation to avoid the Fatal IE JavaScript Bug

One of the most terrible bug you can have on a website is the IE "Operation aborted" error which simply crashes the rendering of the website... client-side! This is due to the bad quality of the rendering engine of IE6 and IE7 which can crash if you append DOM elements to the `body` element from a script which is not directly a child of the `body` element. Unfortunately, this is typically the case if you load the Facebook Connect JavaScript without being careful to load it only directly from the `body` element and at the end of your document. But, this can easily be solved with symfony by using `slots`. Use a `slot` to include the Facebook Connect script whenever necessary in your template and render it in the layout at the end of the document, before the `</body>` tag:

*Listing 12-13*

```
// in a template that uses a XFBML tag or a Facebook Connect button
slot('fb_connect');
include_facebook_connect_script();
end_slot();

// just before </body> in the layout to avoid problems in IE
if (has_slot('fb_connect'))
{
  include_slot('fb_connect');
}
```

## Best Practices for Facebook Applications

Thanks to the `sfFacebookConnectPlugin`, the integration with `sfGuardPlugin` is made seamlessly and the choice of whether the application will be FBML, IFrame or a Facebook Connect website can wait until the last minute. To go further and create a real application using many more Facebook features, here are some important tips that leverage symfony's features.

### Using symfony's Environments to set up multiple Facebook Connect test Servers

A very important aspect of the symfony philosophy is that of fast debugging and quality testing of the application. Using Facebook can really make this difficult since many features

need an Internet connection to communicate with the Facebook server, and an open 80 port to exchange authentication cookies. Additionally, there is another constraint: a Facebook Connect application can only be connected to one host. This is a real problem if the application is developed on a machine, tested on another, put in pre-production on a third server and used finally on a fourth one. In that case the most straightforward solution is to actually create an application for each server and create a symfony environment for each of them. This is very simple in symfony: do a simple copy and paste of the `frontend_dev.php` file or its equivalent into `frontend_preprod.php` and edit the line in the newly created file to change the `dev` environment into the new `preprod` environment:

*Listing 12-14*
```
$configuration =
ProjectConfiguration::getApplicationConfiguration('frontend', 'preprod',
true);
```

Next, edit your `app.yml` file to configure the different Facebook applications corresponding to each environment:

*Listing 12-15*
```
prod:
  facebook:
    api_key: xxx
    api_secret: xxx
    api_id: xxx

dev:
  facebook:
    api_key: xxx
    api_secret: xxx
    api_id: xxx

preprod:
  facebook:
    api_key: xxx
    api_secret: xxx
    api_id: xxx
```

Now the application will be testable on every different server using the corresponding `frontend_xxx.php` entry point.

## Using symfony's logging System for debugging FBML

The layout-switching solution enables development and testing of most of an FBML application outside the Facebook website. However, the final test inside Facebook can sometimes result, nonetheless, in the most obscure error message. Indeed, the main problem of rendering FBML directly in Facebook is the fact that error 500 are caught and replaced by a not-very-helpful standard error message. On top of that, the web debug toolbar, to which symfony developers quickly get addicted, is not rendered in the Facebook frame. Fortunately the very good logging system of symfony is there to save us. The `sfFacebookConnectPlugin` automatically logs many important actions and it is easy to add lines in the logging file at any point in the application:

*Listing 12-16*
```
if (sfConfig::get('sf_logging_enabled'))
{
  sfContext::getInstance()->getLogger()->info($message);
}
```

## Using a Proxy to avoid wrong Facebook Redirections

One strange bug of Facebook is that once Facebook Connect is configured in the application, the Facebook Connect server is considered as home of the application. Although the home can be configured, it has to be in the domain of the Facebook Connect host. So no other solution exists other than to surrender and configure your home to a simple symfony action redirecting to wherever needed. The following code will redirect to the Facebook application:

Listing 12-17

```
public function executeRedirect(sfWebRequest $request)
{

  return
$this->redirect('http://apps.facebook.com'.sfConfig::get('app_facebook_app_url'));
}
```

## Using the `fb_url_for()` helper in Facebook applications

To keep an agnostic application that can be used as FBML in Facebook or XFBML in an IFrame until the last minute, an important problem is the routing:

- For an FBML application, the links inside the application need to point to `/app-name/symfony-route`;
- for an IFrame application, it is important to pass the Facebook session information from page to page.

The `sfFacebookConnectPlugin` provides a special helper which can do both, the `fb_url_for()` helper.

## Redirecting inside an FBML application

Symfony developers quickly become accustomed to redirecting after a successful post, a good practice in web development to avoid double-posting. Redirecting in an FBML application, however, does not work as expected. Instead, a special FBML tag `<fb:redirect>` is needed to tell Facebook to do the redirection. To stay agnostic depending on the context (the FBML tag or the normal symfony redirect) a special redirect function exists in the `sfFacebook` class, which can be used, for example, in a form saving action:

Listing 12-18

```
if ($form->isValid())
{
  $form->save();

  return sfFacebook::redirect($url);
}
```

## Connecting existing Users with their Facebook Account

One of the goals of Facebook Connect is to ease the registration process for new users. However, another interesting use is to also connect existing users with their Facebook account, either to get more information about them or to communicate in their feed. This can be achieved in two ways:

- Push existing sfGuard users to click on the “Connect with Facebook” button. The `sfFacebookConnectAuth/signIn` action will not create a new sfGuard user if it detects a currently logged-in user, but will simply save the newly Facebook Connected user to the current sfGuard user. It is that easy.

- Use the email recognition system of Facebook. When a user uses Facebook Connect on a website, Facebook can provide a special hash of his emails, which can be compared to the email hashes in the existing database to recognize an account belonging to the user which has been created before. However, for security reasons most likely, Facebook only provides these email hashes if the user has previously registered using their API! Therefore, it is important to register all new users' emails regularly to be able to recognize them later on. This is what the `registerUsers` task does, which has been ported to 1.2 by Damien Alexandre. This task should run at least every night to register newly created users, or after a new user is created, using the `registerUsers` method of `sfFacebookConnect`:

*Listing 12-19*
```
sfFacebookConnect::registerUsers(array($sfGuardUser));
```

# Going further

I hope this chapter managed to fulfill its purpose: help you start developing a Facebook application using symfony and explain how to leverage symfony throughout your Facebook development. However, the `sfFacebookConnectPlugin` does not replace the Facebook API, and to learn about using the full power of the Facebook development platform you will have to visit its website[136].

To conclude, I want to thank the symfony community for its quality and generosity, especially those who already contributed to the `sfFacebookConnectPlugin` through their comments and patches: Damien Alexandre, Thomas Parisot, Maxime Picaud, Alban Creton and sorry to the others I might have forgotten. And of course, if you feel there is something missing in the plugin, do not hesitate to contribute yourself!

136. `http://developers.facebook.com/`

Chapter 13

# Leveraging the Power of the Command Line

*by Geoffrey Bachelet*

Symfony 1.1 introduced a modern, powerful, and flexible command line system in replacement of the old pake-based tasks system. From version to version, the tasks system has been improved to make it what it is today.

Many web developers won't see the added value in tasks. Often, those developers don't realize the power of the command line. In this chapter, we are going to dive into tasks, from the very beginning to more advanced usage, seeing how it can help your everyday work, and how you can get the best from tasks.

## Tasks at a Glance

A task is a piece of code that is run from the command line using the `symfony` php script at the root of your project. You may already have run into tasks through the well-known `cache:clear` task (also known as `cc`) by running it in a shell:

```
$ php symfony cc
```

*Listing 13-1*

Symfony provides a set of general purpose built-in tasks for a variety of uses. You can get a list of the available tasks by running the `symfony` script without any arguments or options:

```
$ php symfony
```

*Listing 13-2*

The output will look something like this (content truncated):

```
Usage:
  symfony [options] task_name [arguments]

Options:
  --help        -H  Display this help message.
  --quiet       -q  Do not log messages to standard output.
  --trace       -t  Turn on invoke/execute tracing, enable full backtrace.
  --version     -V  Display the program version.
  --color           Forces ANSI color output.
  --xml             To output help as XML

Available tasks:
  :help                        Displays help for a task (h)
```

*Listing 13-3*

```
  :list                        Lists tasks
app
  :routes                      Displays current routes for an application
cache
  :clear                       Clears the cache (cc, clear-cache)
```

You may have already noticed that tasks are grouped. Groups of tasks are called namespaces, and tasks name are generally composed of a namespace and a task name (except for the `help` and `list` tasks that don't have a namespace). This naming scheme allows for easy task categorization, and you should choose a meaningful namespace for each of your tasks.

# Writing your own Tasks

Getting started writing tasks with symfony takes only a few minutes. All you have to do is create your task, name it, put some logic into it, and voilà, you're ready to run your first custom task. Let's create a very simple *Hello, World!* task, for example in `lib/task/sayHelloTask.class.php`:

*Listing 13-4*

```
class sayHelloTask extends sfBaseTask
{
  public function configure()
  {
    $this->namespace = 'say';
    $this->name      = 'hello';
  }

  public function execute($arguments = array(), $options = array())
  {
    echo 'Hello, World!';
  }
}
```

Now run it with the following command:

*Listing 13-5*

```
$ php symfony say:hello
```

This task will only output *Hello, World!*, but it's only a start! Tasks are not really meant to output content directly through the `echo` or `print` statements. Extending `sfBaseTask` allows us to use a handful of helpful methods, including the `log()` method, which does just what we want to do, output content:

*Listing 13-6*

```
public function execute($arguments = array(), $options = array())
{
  $this->log('Hello, World!');
}
```

Since a single task call can result in multiple tasks outputting content, you may actually want to use the `logSection()` method:

*Listing 13-7*

```
public function execute($arguments = array(), $options = array())
{
  $this->logSection('say', 'Hello, World!');
}
```

Now, you might have already noticed the two arguments passed to the `execute()` method, `$arguments` and `$options`. These are meant to hold all arguments and options passed to

your task at runtime. We will cover arguments and options extensively later. For now, let's just add a bit of interactivity to our task by allowing the user to specify who we want to say hello to:

Listing 13-8
```
public function configure()
{
  $this->addArgument('who', sfCommandArgument::OPTIONAL, 'Who to say hello
to?', 'World');
}

public function execute($arguments = array(), $options = array())
{
  $this->logSection('say', 'Hello, '.$arguments['who'].'!');
}
```

Now the following command:

Listing 13-9
```
$ php symfony say:hello Geoffrey
```

Should produce the following output:

Listing 13-10
```
>> say       Hello, Geoffrey!
```

Wow, that was easy.

By the way, you might want to include a little more metadata in the tasks, like what it does for example. You can do so by setting the `briefDescription` and `detailedDescription` properties:

Listing 13-11
```
public function configure()
{
  $this->namespace           = 'say';
  $this->name                = 'hello';
  $this->briefDescription    = 'Simple hello world';

  $this->detailedDescription = <<<EOF
The [say:hello|INFO] task is an implementation of the classical
Hello World example using symfony's task system.

  [./symfony say:hello|INFO]

Use this task to greet yourself, or somebody else using
the [--who|COMMENT] argument.
EOF;

  $this->addArgument('who', sfCommandArgument::OPTIONAL, 'Who to say hello
to?', 'World');
}
```

As you can see, you can use a basic set of markup to decorate your description. You can check the rendering using symfony's task help system:

Listing 13-12
```
$ php symfony help say:hello
```

# The Options System

Options in a symfony task are organized into two distinct sets, options and arguments.

## Options

Options are those that you pass using hyphens. You can add them to your command line in any order. They can either have a value or not, in which case they act as a boolean. More often than not, options have both a long and short form. The long form is usually invoked using two hyphens while the short form requires only one hyphen.

Examples of common options are the help switch (`--help` or `-h`), the verbosity switch (`--quiet` or `-q`) or the version switch (`--version` or `-V`).

Options are defined with an `sfCommandOption` class and stored in an `sfCommandOptionSet` class.

## Arguments

Arguments are just a piece of data that you append to your command line. They must be specified in the same order in which they were defined, and you must enclose them in quotes if you want to include a space in them (or you could also escape the spaces). They can be either required or optional, in which case you should specify a default value in the argument's definition.

Obviously, arguments are defined with an `sfCommandArgument` class and stored in an `sfCommandArgumentSet` class.

## Default Sets

Every symfony task comes with a set of default options and arguments:

- `--help` (`-H`): Displays this help message.
- `--quiet` (`-q`): Do not log messages to standard output.
- `--trace` (`-t`): Turns on invoke/execute tracing, enable full backtrace.
- `--version` (`-V`): Displays the program version.
- `--color`: Forces ANSI color output.

## Special Options

Symfony's task system understands two very special options, `application` and `env`.

The `application` option is needed when you want access to an `sfApplicationConfiguration` instance rather than just an `sfProjectConfiguration`. instance. This is the case, for example, when you want to generate URLs, since routing is generally associated to a specific application.

When an `application` option is passed to a task, symfony will automatically detect it and create the corresponding `sfApplicationConfiguration` object instead of the default `sfProjectConfiguration` object. Note that you can set a default value for this option, hence saving you the hassle of having to pass an application by hand each time you run the task.

The `env` option controls, obviously, the environment in which the task executes. When no environment is passed, `test` is used by default. Just like for `application`, you can set a default value for the `env` option that will automatically be used by symfony.

Since `application` and `env` are not included in the default options set, you have to add them by hand in your task:

*Listing 13-13*

```
public function configure()
{
  $this->addOptions(array(
    new sfCommandOption('application', null,
sfCommandOption::PARAMETER_REQUIRED, 'The application name', 'frontend'),
    new sfCommandOption('env', null, sfCommandOption::PARAMETER_REQUIRED,
'The environment', 'dev'),
  ));
}
```

In this example, the `frontend` application will be automatically used, and unless a different environment is specified, the task will run in the `dev` environment.

# Accessing the Database

Having access to your database from inside a symfony task is just a matter of instantiating an `sfDatabaseManager` instance:

*Listing 13-14*

```
public function execute($arguments = array(), $options = array())
{
  $databaseManager = new sfDatabaseManager($this->configuration);
}
```

You can also access the ORM's connection object directly:

*Listing 13-15*

```
public function execute($arguments = array(), $options = array())
{
  $databaseManager = new sfDatabaseManager($this->configuration);
  $connection = $databaseManager->getDatabase()->getConnection();
}
```

But what if you have several connections defined in your `databases.yml`? You could, for example, add a `connection` option to your task:

*Listing 13-16*

```
public function configure()
{
  $this->addOption('connection', sfCommandOption::PARAMETER_REQUIRED, 'The
connection name', 'doctrine');
}

public function execute($arguments = array(), $options = array())
{
  $databaseManager = new sfDatabaseManager($this->configuration);
  $connection =
$databaseManager->getDatabase(isset($options['connection']) ?
$options['connection'] : null)->getConnection();
}
```

As usual, you can set a default value for this option.

Voilà! You can now manipulate your models just as if you were in your symfony application.

Be careful when batch processing using your favorite ORM's objects. Both Propel and Doctrine suffer from a well known PHP bug related to cyclic references and the garbage collector that results in a memory leak. This has been partially fixed in PHP 5.3.

# Sending Emails

One of the most common use for tasks is sending emails. Until symfony 1.3, sending email was not really straightforward. But times have changed: symfony now features full integration with Swift Mailer[137], a feature-rich PHP mailer library, so let's use it!

Symfony's task system exposes the mailer object through the `sfCommandApplicationTask::getMailer()` method. That way, you can gain access to the mailer and easily send emails:

*Listing 13-17*
```
public function execute($arguments = array(), $options = array())
{
  $mailer  = $this->getMailer();
  $mailer->composeAndSend($from, $recipient, $subject, $messageBody);
}
```

Since the mailer's configuration is read from the application configuration, your task must accept an application option in order to be able to use the mailer.

If you are using the spool strategy, emails are only sent when you call the `project:send-emails` task.

In most cases, you won't have your message's content sitting in a magical `$messageBody` variable just waiting to be sent, you'll want to somehow generate it. There is no preferred way in symfony to generate content for your emails, but there are a couple tips you can follow to make your life easier:

## Delegate Content Generation

For example, create a protected method for your task that returns the content for the email you're sending:

*Listing 13-18*
```
public function execute($arguments = array(), $options = array())
{
  $this->getMailer()->composeAndsend($from, $recipient, $subject,
$this->getMessageBody());
}

protected function getMessageBody()
{
  return 'Hello, World';
}
```

137. http://swiftmailer.org/

## Use Swift Mailer's Decorator Plugin

Swift Mailer features a plugin known as `Decorator`[138] that is basically a very simple, yet efficient, template engine that can take recipient-specific replacement value-pairs and apply them throughout all mails being sent.

See Swift Mailer's documentation[139] for more information.

## Use an external Templating Library

Integrating a third party templating library is easy. For example, you could use the brand new templating component released as part of the Symfony Components project. Just drop the component code somewhere in your project (`lib/vendor/templating/` would be a good place), and put down the following code in your task:

*Listing 13-19*

```
protected function getMessageBody($template, $vars = array())
{
  $engine = $this->getTemplateEngine();
  return $engine->render($template, $vars);
}

protected function getTemplateEngine()
{
  if (is_null($this->templateEngine))
  {
    $loader = new sfTemplateLoaderFilesystem(sfConfig::get('sf_app_dir').'/
templates/emails/%s.php');
    $this->templateEngine = new sfTemplateEngine($loader);
  }

  return $this->templateEngine;
}
```

## Getting the best of both Worlds

There's still more that you can do. Swift Mailer's `Decorator` plugin is very handy since it can manage replacements on a recipient-specific basis. It means that you define a set of replacements for each of your recipients, and Swift Mailer takes care of replacing tokens with the right value based on the recipient of the mail being sent. Let's see how we can integrate this with the templating component:

*Listing 13-20*

```
public function execute($arguments = array(), $options = array())
{
  $message = Swift_Message::newInstance();

  // fetches a list of users
  foreach($users as $user)
  {
    $replacements[$user->getEmail()] = array(
      '{username}'      => $user->getEmail(),
      '{specific_data}' => $user->getSomeUserSpecificData(),
    );

    $message->addTo($user->getEmail());
```

138. http://swiftmailer.org/docs/decorator-plugin
139. http://swiftmailer.org/docs/

```
    }

    $this->registerDecorator($replacements);

    $message
      ->setSubject('User specific data for {username}!')
      ->setBody($this->getMessageBody('user_specific_data'));

    $this->getMailer()->send($message);
  }

  protected function registerDecorator($replacements)
  {
    $this->getMailer()->registerPlugin(new
Swift_Plugins_DecoratorPlugin($replacements));
  }

  protected function getMessageBody($template, $vars = array())
  {
    $engine = $this->getTemplateEngine();
    return $engine->render($template, $vars);
  }

  protected function getTemplateEngine($replacements = array())
  {
    if (is_null($this->templateEngine))
    {
      $loader = new
sfTemplateLoaderFilesystem(sfConfig::get('sf_app_template_dir').'/emails/
%s.php');
      $this->templateEngine = new sfTemplateEngine($loader);
    }

    return $this->templateEngine;
  }
```

With `apps/frontend/templates/emails/user_specific_data.php` containing the following code:

*Listing 13-21*

```
Hi {username}!

We just wanted to let you know your specific data:

{specific_data}
```

And that's it! You now have a fully featured template engine to build your email content.

# Generating URLs

Writing emails usually requires that you generate URLs based on your routing configuration. Fortunately enough, generating URLs has been made easy in symfony 1.3 since you can directly access the routing of the current application from inside a task by using the `sfCommandApplicationTask::getRouting()` method:

*Listing 13-22*

```
public function execute($arguments = array(), $options = array())
{
```

```
  $routing = $this->getRouting();
}
```

Since the routing is application dependent, you have to make sure that your application has an application configuration available, otherwise you won't be able to generate URLs using the routing.

See the *Special Options* section to learn how to automatically get an application configuration in your task.

Now that we have a routing instance, it's quite straightforward to generate a URL using the `generate()` method:

Listing 13-23

```
public function execute($arguments = array(), $options = array())
{
  $url = $this->getRouting()->generate('default', array('module' => 'foo',
'action' => 'bar'));
}
```

The first argument is the route's name and the second is an array of parameters for the route. At this point, we have generated a relative URL, which is most likely not what we want. Unfortunately, generating absolute URLs in a task will not work since we don't have an `sfWebRequest` object to rely on to fetch the HTTP host.

One simple way to solve this is to set the HTTP host in your `factories.yml` configuration file:

Listing 13-24

```
all:
  routing:
    class: sfPatternRouting
    param:
      generate_shortest_url:            true
      extra_parameters_as_query_string: true
      context:
        host: example.org
```

See the `context_host` setting? This is what the routing will use when asked for an absolute URL:

Listing 13-25

```
public function execute($arguments = array(), $options = array())
{
  $url = $this->getRouting()->generate('my_route', array(), true);
}
```

# Accessing the I18N System

Not all factories are as easily accessible as the mailer and the routing. Should you need access to one of them, it's really not too hard to instantiate them. For example, say you want to internationalize your tasks, you would then want to access symfony's i18n subsystem. This is easily done using the `sfFactoryConfigHandler`:

Listing 13-26

```
protected function getI18N($culture = 'en')
{
  if (!$this->i18n)
  {
    $config =
```

```
sfFactoryConfigHandler::getConfiguration($this->configuration->getConfigPaths('config/
factories.yml'));
    $class  = $config['i18n']['class'];

    $this->i18n = new $class($this->configuration, null,
$config['i18n']['param']);
  }

  $this->i18n->setCulture($culture);

  return $this->i18n;
}
```

Let's see what's going on here. First, we are using a simple caching technique to avoid re-building the i18n component at each call. Then, using the `sfFactoryConfigHandler`, we retrieve the component's configuration in order to instantiate it. We finish by setting the culture configuration. The task now has access to internationalization:

*Listing 13-27*

```
public function execute($arguments = array(), $options = array())
{
  $this->log($this->getI18N('fr')->__('some translated text!'));
}
```

Of course, always passing the culture is not very handy, especially if you don't need to change culture very often in your task. We will see how to arrange that in the next section.

# Refactoring your Tasks

Since sending emails (and generating content for them) and generating URLs are two very common task, it may be a good idea to create a base task that provides these two features automatically for each task. This is fairly easy to do. Create a base class inside your project, for example `lib/task/sfBaseEmailTask.class.php`.

*Listing 13-28*

```
class sfBaseEmailTask extends sfBaseTask
{
  protected function registerDecorator($replacements)
  {
    $this->getMailer()->registerPlugin(new
Swift_Plugins_DecoratorPlugin($replacements));
  }

  protected function getMessageBody($template, $vars = array())
  {
    $engine = $this->getTemplateEngine();
    return $engine->render($template, $vars);
  }

  protected function getTemplateEngine($replacements = array())
  {
    if (is_null($this->templateEngine))
    {
      $loader = new
sfTemplateLoaderFilesystem(sfConfig::get('sf_app_template_dir').'/
templates/emails/%s.php');
      $this->templateEngine = new sfTemplateEngine($loader);
    }
```

```
    return $this->templateEngine;
  }
}
```

While we're at it, we are going to automate the task's options setup. Add these methods to the `sfBaseEmailTask` class:

*Listing 13-29*

```
public function configure()
{
  $this->addOption('application', null,
sfCommandOption::PARAMETER_REQUIRED, 'The application', 'frontend');
}

protected function generateUrl($route, $params = array())
{
  return $this->getRouting()->generate($route, $params, true);
}
```

We use the `configure()` method to add common options to all extending tasks. Unfortunately, any class extending `sfBaseEmailTask` will now have to call `parent::configure` in its own `configure()` method, but that's really a minor annoyance in regard of added value.

Now let's refactor the I18N access code from the previous section:

*Listing 13-30*

```
public function configure()
{
  $this->addOption('application', null,
sfCommandOption::PARAMETER_REQUIRED, 'The application', 'frontend');
  $this->addOption('culture', null, sfCommandOption::PARAMETER_REQUIRED,
'The culture', 'en');
}

protected function getI18N()
{
  if (!$this->i18n)
  {
    $config =
sfFactoryConfigHandler::getConfiguration($this->configuration->getConfigPaths('config/
factories.yml'));
    $class  = $config['i18n']['class'];

    $this->i18n = new $class($this->configuration, null,
$config['i18n']['param']);


$this->i18n->setCulture($this->commandManager->getOptionValue('culture'));
  }

  return $this->i18n;
}

protected function changeCulture($culture)
{
  $this->getI18N()->setCulture($culture);
}
```

```
protected function process(sfCommandManager $commandManager, $options)
{
  parent::process($commandManager, $options);
  $this->commandManager = $commandManager;
}
```

We have a problem to solve here: it is not possible to access arguments and options values outside `execute()`'s scope. To fix that, we are simply overloading the `process()` method to attach the options manager to the class. The options manager is, as its name says, managing arguments and options for the current task. For example, you can access options values via the `getOptionValue()` method.

# Executing a Task inside a Task

An alternative way to refactor your tasks is to embed a task inside another task. This is made particularly easy through the `sfCommandApplicationTask::createTask()` and `sfCommandApplicationTask::runTask()` methods.

The `createTask()` method will create an instance of a task for you. Just pass it a task name, just as if you were on the command line, and it will return you an instance of the desired task, ready to be run:

*Listing 13-31*
```
$task = $this->createTask('cache:clear');
$task->run();
```

But since we are lazy, the `runTask` does everything for us:

*Listing 13-32*
```
$this->runTask('cache:clear');
```

Of course, you can pass arguments and options (in this order):

*Listing 13-33*
```
$this->runTask('plugin:install', array('sfGuardPlugin'),
array('install_deps' => true));
```

Embedding tasks is useful for composing powerful tasks from more simple tasks. For example, you could combine several tasks in a `project:clean` task that you would run after each deployment:

*Listing 13-34*
```
$tasks = array(
  'cache:clear',
  'project:permissions',
  'log:rotate',
  'plugin:publish-assets',
  'doctrine:build-model',
  'doctrine:build-forms',
  'doctrine:build-filters',
  'project:optimize',
  'project:enable',
);

foreach($tasks as $task)
{
  $this->run($task);
}
```

# Manipulating the Filesystem

Symfony comes with a built-in simple filesystem abstraction (`sfFilesystem`) that permits the execution of simple operations on files and directories. It is accessible inside a task with `$this->getFilesystem()`. This abstraction exposes the following methods:

- `sfFilesystem::copy()`, to copy a file
- `sfFilesystem::mkdirs()`, creates recursive directories
- `sfFilesystem::touch()`, to create a file
- `sfFilesystem::remove()`, to delete a file or directory
- `sfFilesystem::chmod()`, to change permissions on a file or directory
- `sfFilesystem::rename()`, to rename a file or directory
- `sfFilesystem::symlink()`, to create a link to a directory
- `sfFilesystem::relativeSymlink()`, to create a relative link to a directory
- `sfFilesystem::mirror()`, to mirror a complete file tree
- `sfFilesystem::execute()`, to execute an arbitrary shell command

It also exposes a very handy method that we are going to cover in the next section: `replaceTokens()`.

# Using Skeletons to generate Files

Another common use for tasks is to generate files. Generating files can be made easy using skeletons and the aforementioned method `sfFilesystem::replaceTokens()`. As its name suggests, this methods replaces tokens inside a set of files. That is, you pass it an array of file, a list of tokens and it replaces every occurrence of each token with its assigned value, for each file in the array.

To better understand how this is useful, we are going to partially rewrite an existing task: `generate:module`. For the sake of clarity and brevity, we will only look at the `execute` part of this task, assuming it has been configured properly with all needed options. We will also skip validation.

Even before starting to write the task, we need to create a skeleton for the directories and files we are going to create, and store it somewhere like `data/skeleton/`:

*Listing 13-35*

```
data/skeleton/
  module/
    actions/
      actions.class.php
    templates/
```

The `actions.class.php` skeleton could look like something like this:

*Listing 13-36*

```
class %moduleName%Actions extends %baseActionsClass%
{
}
```

The first step of our task will be to mirror the file tree at the right place:

*Listing 13-37*

```
$moduleDir = sfConfig::get('sf_app_module_dir').$options['module'];
$finder    = sfFinder::type('any');
$this->getFilesystem()->mirror(sfConfig::get('sf_data_dir').'/skeleton/
module', $moduleDir, $finder);
```

Now let's replace the tokens in `actions.class.php`:

Listing 13-38
```
$tokens = array(
  'moduleName'       => $options['module'],
  'baseActionsClass' => $options['base-class'],
);

$finder = sfFinder::type('file');
$this->getFilesystem()->replaceTokens($finder->in($moduleDir), '%', '%',
$tokens);
```

And that's it, we generated our new module, using token replacing to customize it.

The built-in `generate:module` actually looks into `data/skeleton/` for alternative skeleton to use instead of the default ones, so watch your step!

## Using a dry-run Option

Often you want to be able to preview the result of a task before actually running it. Here are a couple of tips on how to do so.

First, you should use a standard name, such as `dry-run`. Everyone will recognize this for what it is. Until symfony 1.3, `sfCommandApplication` *did* add a default `dry-run` option, but now it should be added by hand (possibly in a base class, as demonstrated above):

Listing 13-39
```
$this->addOption(new sfCommandOption('dry-run', null,
sfCommandOption::PARAMETER_NONE, 'Executes a dry run');
```

You would then invoke your task like this:

Listing 13-40
```
./symfony my:task --dry-run
```

The `dry-run` option indicates that the task should not make any change.

*Should not make any change*, remember this, they are the key words. When running in `dry-run` mode, your task must leave the environment exactly as it was before, including (but not limited to):

- The database: do not insert, update or delete records from your tables. You can use a transaction to achieve this.
- The filesystem: do not create, modify or delete files from your filesystem.
- Email sending: do not send emails, or send them to a debug address.

Here is a simple example of using the `dry-run` option:

Listing 13-41
```
$connection->beginTransaction();

// modify your database

if ($options['dry-run'])
{
  $connection->rollBack();
}
else
{
  $connection->commit();
}
```

# Writing unit Tests

Since tasks can achieve a variety of goals, unit testing them is not an easy thing. As such, there's not one way to test tasks, but there are some principles to follow that can help make your tasks more testable.

First, think of your task like a controller. Remember the rule about controller? *Thin controllers, fat models*. That is, move all the business logic inside your models, that way, you can test your models instead of the task, which is way easier.

Once you think you can't get more logic into models, split your `execute()` method into chunks of easily testable code, each residing in its own easily accessible (read: public) method. Splitting your code has several benefits:

1. it makes your task's `execute` more readable
2. it makes your task more testable
3. it makes your task more extendable

Be creative, don't hesitate to build a small specific environment for your testing needs. And if you can't find any way to test that awesome task that you just wrote, there are two possibilities: either you wrote it bad or you should ask someone for his opinion. Also, you can always dig into someone else's code to see how they test things (symfony's tasks are well tested for example, even generators).

# Helper Methods: Logging

Symfony's task system tries hard to make the developer's day easier, providing handy helper method for common operations such as logging and user interaction.

One can easily log messages to `STDOUT` using the `log` family of methods:

- `log`, accepts an array of messages
- `logSection`, a bit more elaborate, formats your message with a prefix (first argument) and a message type (fourth argument). When you log something too long, like a file path, `logSection` will usually shrink your message, which can prove annoying. Use the third argument to specify a message max size that fits your message
- `logBlock`, is the logging style used for exceptions. Here again, you can pass a formatting style

Available logging formats are `ERROR`, `INFO`, `COMMENT` and `QUESTION`. Don't hesitate to try them to see what they look like.

Example usage:

*Listing 13-42*

```
$this->logSection('file+', $aVeryLongFileName,
$this->strlen($aVeryLongFileName));

$this->logBlock('Congratulations! You ran the task successfuly!', 'INFO');
```

# Helper Methods: User Interaction

Three more helpers are provided to ease user interaction:

- `ask()`, basically prints a question and returns any user input
- `askConfirmation()`, we ask the user for a confirmation, allowing `y` (yes) and `n` (no) as user input

- `askAndValidate()`, a very useful method that prints a question and validates the user input through an `sfValidator` passed as the second argument. The third argument is an array of options in which you can pass a default value (`value`), a maximum number of attempts (`attempts`) and a formatting style (`style`).

For example, you can ask a user for his email address and validate it on the fly:

*Listing 13-43*
```
$email = $this->askAndValidate('What is your email address?', new
sfValidatorEmail());
```

# Bonus Round: Using Tasks with a Crontab

Most UNIX and GNU/Linux systems allows for task planning through a mechanism known as *cron*. The *cron* checks a configuration file (a *crontab*) for commands to run at a certain time. Symfony tasks can easily be integrated into a crontab, and the `project:send-emails` task is a perfect candidate for an example of that:

*Listing 13-44*
```
MAILTO="you@example.org"
0 3 * * *       /usr/bin/php /var/www/yourproject/symfony
project:send-emails
```

This configuration tells *cron* to run the `project:send-emails` every day at 3am and to send all possible output (that is, logs, errors, etc) to the address *you@example.org*.

For more information on the crontab configuration file format, type `man 5 crontab` in a terminal.

You can, and should actually, pass arguments and options:

*Listing 13-45*
```
MAILTO="you@example.org"
0 3 * * *       /usr/bin/php /var/www/yourproject/symfony
project:send-emails --env=prod --application=frontend
```

You should replace `/usr/bin/php` with the location of your PHP CLI binary. If you don't have this information, you can try `which php` on linux systems or `whereis php` on most other UNIX systems.

# Bonus Round: Using STDIN

Since tasks are run in a command line environment, you can access the standard input stream (STDIN). The UNIX command line allows applications to interact between each other by a variety of means, one of which is the *pipe*, symbolized by the character |. The *pipe* allows you to pass an application's output (know as *STDOUT*) to another application's standard input (known as *STDIN*). These are made accessible in your tasks through PHP's special constants `STDIN` and `STDOUT`. There's also a third standard stream, *STDERR*, accessible through `STDERR`, meant to carry an applications' error messages.

So what can we do exactly with the standard input? Well, imagine you have an application running on your server that would like to communicate with your symfony application. You could of course have it communicate through HTTP, but a more efficient way would be to pipe its output to a symfony task. Say the application can send structured data (for example a PHP

array serialization) describing domain objects that you want to include into your database. You could write the following task:

Listing 13-46

```
while ($content = trim(fgets(STDIN)))
{
  if ($data = unserialize($content) !== false)
  {
    $object = new Object();
    $object->fromArray($data);
    $object->save();
  }
}
```

You would then use it like this:

Listing 13-47

```
/usr/bin/data_provider | ./symfony data:import
```

`data_provider` being the application providing new domain objects, and `data:import` being the task you just wrote.

# Final Thoughts

What tasks can achieve is limited only by your imagination. Symfony's task system is powerful and flexible enough that you can do merely anything you can think off. Add to that the power of an UNIX shell, and you are really going to love tasks.

Chapter 14

# Playing with symfony's Config Cache

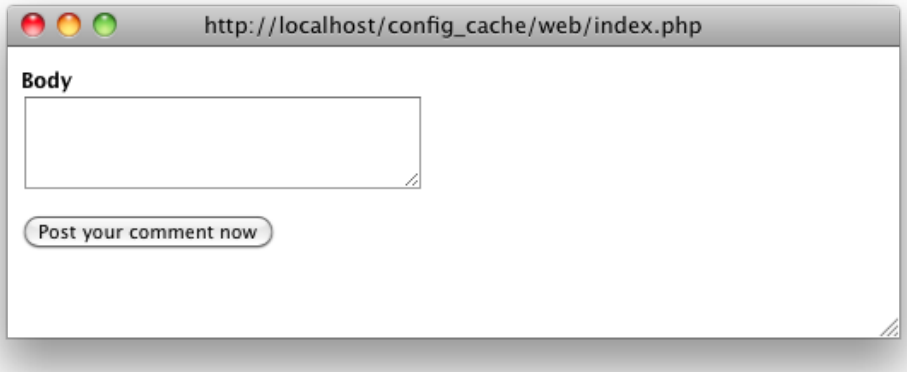
*by Kris Wallsmith*

One of my personal goals as a symfony developer is to streamline each of my peer's workflow as much as possible on any given project. While I may know our codebase inside and out, that's not a reasonable expectation for everyone on the team. Thankfully, symfony provides mechanisms for isolating and centralizing functionality within a project, making it easy for others to make changes with a very light footprint.

## Form Strings

An excellent example of this is the symfony form framework. The form framework is a powerful component of symfony that gives you great control over your forms by moving their rendering and validation into PHP objects. This is a godsend for the application developer, because it means you can encapsulate complex logic in a single form class and extend and reuse it in multiple places.

However, from a template developer's perspective, this abstraction of how a form renders can be troublesome. Take a look at the following form:




The class that configures this form looks something like this:

*Listing 14-1*

```
// lib/form/CommentForm.class.php
class CommentForm extends BaseForm
{
  public function configure()
  {
    $this->setWidget('body', new sfWidgetFormTextarea());
```

```
    $this->setValidator('body', new sfValidatorString(array(
      'min_length' => 12,
    )));
  }
}
```

The form is then rendered in a PHP template like this:

Listing 14-2

```
<!-- apps/frontend/modules/main/templates/indexSuccess.php -->
<form action="#" method="post">
  <ul>
    <li>
      <?php echo $form['body']->renderLabel() ?>
      <?php echo $form['body'] ?>
      <?php echo $form['body']->renderError() ?>
    </li>
  </ul>
  <p><button type="submit">Post your comment now</button></p>
</form>
```

The template developer has quite a bit of control over how this form is rendered. He can change the default labels to be a bit more friendly:

Listing 14-3

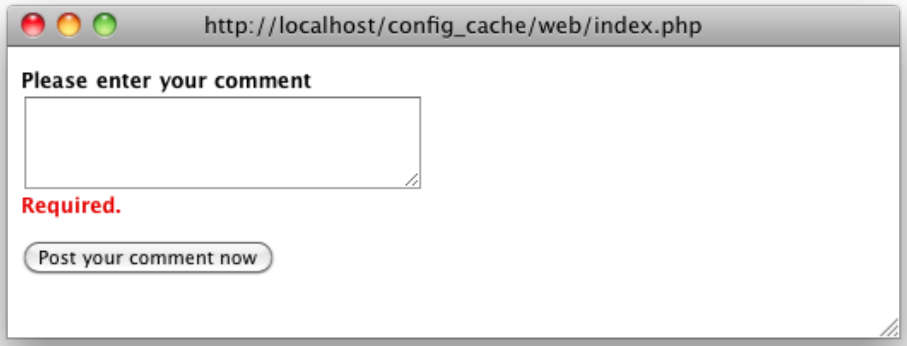
```
<?php echo $form['body']->renderLabel('Please enter your comment') ?>
```

He can add a class to the input fields:

Listing 14-4

```
<?php echo $form['body']->render(array('class' => 'comment')) ?>
```

These modifications are intuitive and easy. But what if he needs to modify an error message?




The `->renderError()` method does not accept any arguments, so the template developer's only recourse is to open the form's class file, find the code that creates the validator in question, and modify its constructor so the new error messages are associated with the appropriate error codes.

In our example, the template developer would have to make the following change:

Listing 14-5

```
// before
$this->setValidator('body', new sfValidatorString(array(
  'min_length' => 12,
)));

// after
```

```
$this->setValidator('body', new sfValidatorString(array(
  'min_length' => 12,
), array(
  'min_length' => 'You haven't written enough',
)));
```

Notice a problem? Oops! I used an apostrophe inside a single-quoted string. Of course you or I would never make such a silly mistake, but what's to say a template developer mucking around inside a form class won't?

In all seriousness, can we expect template developers to know their way around the symfony form framework well enough to pinpoint exactly where an error message is defined? Should someone working in the view layer be expected to know the signature for a validator's constructor?

I'm pretty sure we can all agree that the answer to these questions is no. Template developers do a lot of valuable work but it's simply unreasonable to expect someone who isn't writing application code to learn the inner-workings of the symfony form framework.

# YAML: A Solution

To simplify the process of editing form strings we are going to add a layer of YAML configuration that enhances each form object as it's passed to the view. This configuration file will look something like this:

*Listing 14-6*

```
# config/forms.yml
CommentForm:
  body:
    label:        Please enter your comment
    attributes:   { class: comment }
    errors:
      min_length: You haven't written enough
```

This is a lot easier, right? The configuration explains itself, plus the apostrophe issue we encountered earlier is now moot. So let's build it!

# Filtering Template Variables

The first challenge is to find a hook in symfony that will allow us to filter every form variable passed to a template through this configuration. To do this, we use the `template.filter_parameters` event, which is fired from the symfony core just prior to rendering a template or template partial.

*Listing 14-7*

```
// lib/form/sfFormYamlEnhancer.class.php
class sfFormYamlEnhancer
{
  public function connect(sfEventDispatcher $dispatcher)
  {
    $dispatcher->connect('template.filter_parameters',
      array($this, 'filterParameters'));
  }

  public function filterParameters(sfEvent $event, $parameters)
  {
    foreach ($parameters as $name => $param)
```

```
    {
      if ($param instanceof sfForm && !$param->getOption('is_enhanced'))
      {
        $this->enhance($param);
        $param->setOption('is_enhanced', true);
      }
    }

    return $parameters;
  }

  public function enhance(sfForm $form)
  {
    // ...
  }
}
```

Notice this code checks an `is_enhanced` option on each form object before enhancing it. This is to prevent forms passed from templates to partials from being enhanced twice.

This enhancer class needs to be connected from your application configuration:

*Listing 14-8*

```
// apps/frontend/config/frontendConfiguration.class.php
class frontendConfiguration extends sfApplicationConfiguration
{
  public function initialize()
  {
    $enhancer = new sfFormYamlEnhancer($this->getConfigCache());
    $enhancer->connect($this->dispatcher);
  }
}
```

Now that we're able to isolate form variables just before they're passed to a template or partial we have everything we need to make this work. The final task is to apply what's been configured in the YAML.

# Applying the YAML

The easiest way to apply this YAML configuration to each form is to load it into an array and loop through each configuration:

*Listing 14-9*

```
public function enhance(sfForm $form)
{
  $config = sfYaml::load(sfConfig::get('sf_config_dir').'/forms.yml');

  foreach ($config as $class => $fieldConfigs)
  {
    if ($form instanceof $class)
    {
      foreach ($fieldConfigs as $fieldName => $fieldConfig)
      {
        if (isset($form[$fieldName]))
        {
          if (isset($fieldConfig['label']))
          {
```

```
            $form->getWidget($fieldName)->setLabel($fieldConfig['label']);
          }

          if (isset($fieldConfig['attributes']))
          {
            $form->getWidget($fieldName)->setAttributes(array_merge(
              $form->getWidget($fieldName)->getAttributes(),
              $fieldConfig['attributes']
            ));
          }

          if (isset($fieldConfig['errors']))
          {
            foreach ($fieldConfig['errors'] as $code => $msg)
            {
              $form->getValidator($fieldName)->setMessage($code, $msg);
            }
          }
        }
      }
    }
  }
}
```

There are a number of problems with this implementation. First, the YAML file is read from the filesystem and loaded into `sfYaml` every time a form is enhanced. Reading from the filesystem in this fashion should be avoided. Second, there are multiple levels of nested loops and a number of conditionals that will only slow your application down. The solution for both of these problems lies in symfony's config cache.

# The Config Cache

The config cache is composed of a collection of classes that optimize the use of YAML configuration files by automating their translation into PHP code and writing that code to the cache directory for execution. This mechanism will eliminate the overhead necessary to load the contents of our configuration file into `sfYaml` before applying its values.

Let's implement a config cache for our form enhancer. Instead of loading `forms.yml` into `sfYaml`, let's ask the current application's config cache for a pre-processed version.

To do this the `sfFormYamlEnhancer` class will need access to the current application's config cache, so we'll add that to the constructor.

*Listing 14-10*

```
class sfFormYamlEnhancer
{
  protected
    $configCache = null;

  public function __construct(sfConfigCache $configCache)
  {
    $this->configCache = $configCache;
    $this->configCache->registerConfigHandler('config/forms.yml',
      'sfSimpleYamlConfigHandler');
  }

  // ...
}
```

The custom configuration handler could be registered either in the constructor of the form enhancer class as above or in a config_handlers.yml file located in one of your config directory:

Listing 14-11

```
# config/config_handlers.yml
config/forms.yml
  class: sfSimpleYamlConfigHandler
  file: %SF_ROOT_DIR%/lib/config/sfSimpleYamlConfigHandler.class.php
```

You must specify the full path to your custom configuration handler source files under the `file` entry as the configuration is initialized even before the autoloading.

The config cache needs to be told what to do when a certain configuration file is requested by the application. For now we've instructed the config cache to use `sfSimpleYamlConfigHandler` to process `forms.yml`. This config handler simply parses YAML into an array and caches it as PHP code.

With the config cache in place and a config handler registered for `forms.yml` we can now call it instead of `sfYaml`:

Listing 14-12

```
public function enhance(sfForm $form)
{
  $config = include $this->configCache->checkConfig('config/forms.yml');

  // ...
}
```

This is much better. Not only have we eliminated the overhead of parsing YAML on all but the first request, we've also switched to using `include`, which exposes this read to the boons of op-code caching.

**Development vs. Production environments**

The internals of `->checkConfig()` differ depending on whether your application's debug mode is on or off. In your `prod` environment, when debug mode is off, this method functions as described here:

- Check for a cached version of the requested file
  - If if exists, return the path to that cached file
  - If it doesn't exist:
    - Process the configuration file
    - Save the resulting code to the cache
    - Return the path to the newly cached file

This method functions differently when debug mode is on. Because config files are edited during the course of development, `->checkConfig()` will compare when the original and cached files were last modified to make sure it gets the latest version. This adds a few more steps to how the same method functions when debug mode is off:

- Check for a cached version of the requested file
  - If it doesn't exist:
    - Process the configuration file
    - Save the resulting code to the cache
  - If it exists:
    - Compare when the config and cached files were last modified
    - If the config file was modified most recently:
    - Process the configuration file
    - Save the resulting code to the cache
- Return the path to the cached file

# Cover me, I'm goin' in!

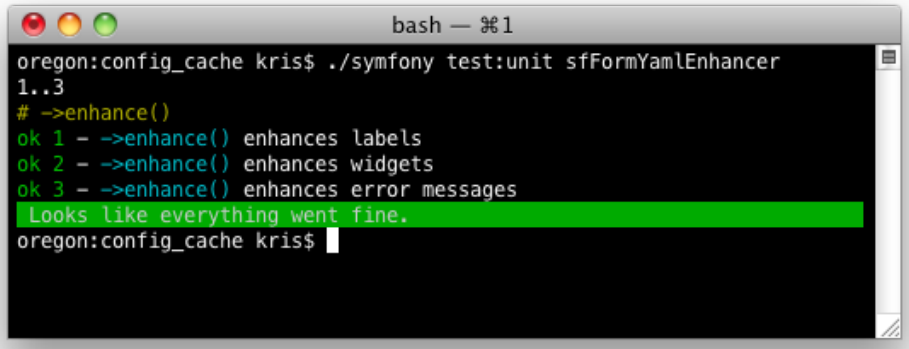
Let's write some tests before going any further. We can start with this basic script:

*Listing 14-13*

```
// test/unit/form/sfFormYamlEnhancerTest.php
include dirname(__FILE__).'/../../bootstrap/unit.php';

$t = new lime_test(3);

$configuration = $configuration->getApplicationConfiguration(
  'frontend', 'test', true, null, $configuration->getEventDispatcher());
sfToolkit::clearDirectory(sfConfig::get('sf_app_cache_dir'));

$enhancer = new sfFormYamlEnhancer($configuration->getConfigCache());

// ->enhance()
$t->diag('->enhance()');

$form = new CommentForm();
$form->bind(array('body' => '+1'));
```

```
$enhancer->enhance($form);

$t->like($form['body']->renderLabel(), '/Please enter your comment/',
  '->enhance() enhances labels');
$t->like($form['body']->render(), '/class="comment"/',
  '->enhance() enhances widgets');
$t->like($form['body']->renderError(), '/You haven\'t written enough/',
  '->enhance() enhances error messages');
```

Running this test against the current `sfFormYamlEnhancer` verifies that it is working correctly:




Now we can go about refactoring with the confidence that our tests will raise a stink if we break anything.

# Custom Config Handlers

In the enhancer code above, every form variable passed to a template will loop through every form class configured in `forms.yml`. This gets the job done, but if you pass multiple form objects to a template, or have a long list of forms configured in the YAML, you may begin to see an impact on performance. This is a good opportunity to write a custom config handler that optimizes this process.

**Why go custom?**

Writing a custom config handler is not for the faint of heart. As with any code generation, config handlers can be error-prone and difficult to test, but the benefits can be plentiful. Creating "hard-coded" logic on-the-fly hits a sweet spot that gives you the advantage of YAML's flexibility and the low-overhead of native PHP code. With an op-code cache added to the mix (such as APC[140] or XCache[141]) config handlers are hard to beat for ease of use and performance.

Most of the magic of config handlers happens behind the scenes. The config cache takes care of the caching logic before it runs any particular config handler so we can just focus on generating the code necessary to apply the YAML configuration.

Each config handler must implement the following two methods:

140. http://pecl.php.net/apc
141. http://xcache.lighttpd.net/

- `static public function getConfiguration(array $configFiles)`
- `public function execute($configFiles)`

The first method, `::getConfiguration()`, is passed an array of file paths, parses them and merges their contents into a single value. In the `sfSimpleYamlConfigHandler` class we used above, this method includes only one line:

*Listing 14-14*
```
static public function getConfiguration(array $configFiles)
{
  return self::parseYamls($configFiles);
}
```

The `sfSimpleYamlConfigHandler` class extends the abstract `sfYamlConfigHandler` which includes a number of helper methods for processing YAML configuration files:

- `::parseYamls($configFiles)`
- `::parseYaml($configFile)`
- `::flattenConfiguration($config)`
- `::flattenConfigurationWithEnvironment($config)`

The first two methods implement symfony's configuration cascade[142]. The second two implement environment-awareness[143].

The `::getConfiguration()` method in our config handler will need a custom method for merging the configuration based on class inheritance. Create an `::applyInheritance()` method that encapsulates this logic:

*Listing 14-15*
```
// lib/config/sfFormYamlEnhancementsConfigHandler.class.php
class sfFormYamlEnhancementsConfigHandler extends sfYamlConfigHandler
{
  public function execute($configFiles)
  {
    $config = self::getConfiguration($configFiles);

    // compile data
    $retval = "<?php\n".
              "// auto-generated by %s\n".
              "// date: %s\nreturn %s;\n";
    $retval = sprintf($retval, __CLASS__, date('Y/m/d H:i:s'),
      var_export($config, true));

    return $retval;
  }

  static public function getConfiguration(array $configFiles)
  {
    return self::applyInheritance(self::parseYamls($configFiles));
  }

  static public function applyInheritance($config)
  {
    $classes = array_keys($config);

    $merged = array();
```

142. http://www.symfony-project.org/reference/1_2/en/03-Configuration-Files-Principles#chapter_03_configuration_cascade
143. http://www.symfony-project.org/reference/1_2/en/03-Configuration-Files-Principles#chapter_03_environment_awareness

```
    foreach ($classes as $class)
    {
      if (class_exists($class))
      {
        $merged[$class] = $config[$class];
        foreach (array_intersect(class_parents($class), $classes) as
$parent)
        {
          $merged[$class] = sfToolkit::arrayDeepMerge(
            $config[$parent],
            $merged[$class]
          );
        }
      }
    }

    return $merged;
  }
}
```

We now have an array whose values have been merged per class inheritance. This eliminates the need to filter the entire configuration through `instanceof` for each form object. What's more, this merge is done in the config handler so it will only happen once and then be cached.

Now we can apply this merged array to a form object with a simple bit of search logic:

*Listing 14-16*

```
class sfFormYamlEnhancer
{
  protected
    $configCache = null;

  public function __construct(sfConfigCache $configCache)
  {
    $this->configCache = $configCache;
    $this->configCache->registerConfigHandler('config/forms.yml',
      'sfFormYamlEnhancementsConfigHandler');
  }

  // ...

  public function enhance(sfForm $form)
  {
    $config = include $this->configCache->checkConfig('config/forms.yml');

    $class = get_class($form);
    if (isset($config[$class]))
    {
      $fieldConfigs = $config[$class];
    }
    else if ($overlap = array_intersect(class_parents($class),
      array_keys($config)))
    {
      $fieldConfigs = $config[current($overlap)];
    }
    else
    {
      return;
    }
```

```
    foreach ($fieldConfigs as $fieldName => $fieldConfig)
    {
      // ...
    }
  }
}
```

Before we run the test script again, let's add an assertion for the new class inheritance logic.

Listing 14-17

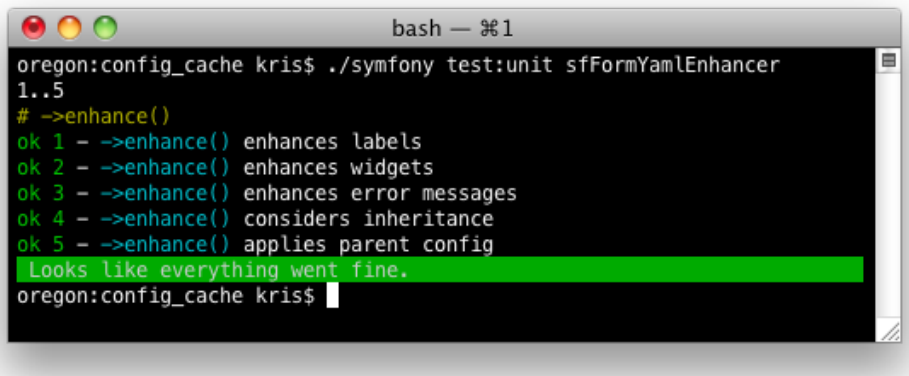
```
# config/forms.yml

# ...

BaseForm:
  body:
    errors:
      min_length: A base min_length message
      required:   A base required message
```

We can verify that the new `required` message is being applied in the test script, and confirm that child forms will receive their parents' enhancements, even if there are none configured for the child class.

Listing 14-18

```
$t = new lime_test(5);

// ...

$form = new CommentForm();
$form->bind();
$enhancer->enhance($form);
$t->like($form['body']->renderError(), '/A base required message/',
  '->enhance() considers inheritance');

class SpecialCommentForm extends CommentForm { }
$form = new SpecialCommentForm();
$form->bind();
$enhancer->enhance($form);
$t->like($form['body']->renderLabel(), '/Please enter your comment/',
  '->enhance() applies parent config');
```

Run this updated test script to verify the form enhancer is working as expected.



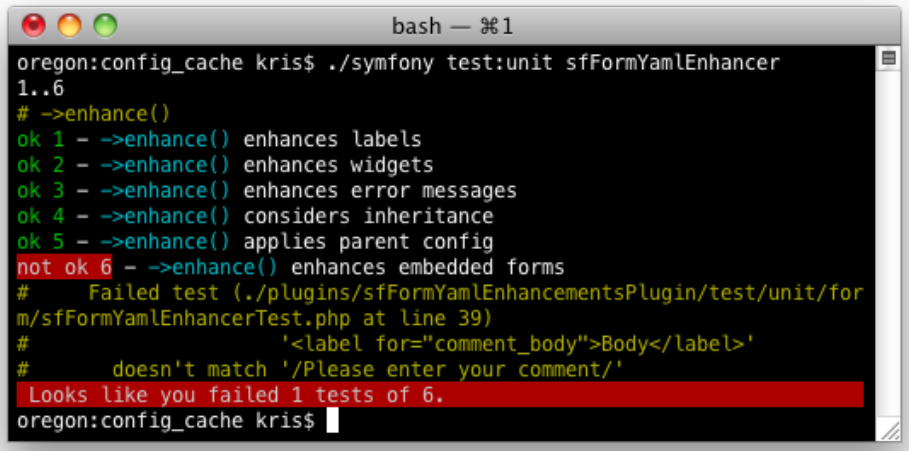
# Getting Fancy with Embedded Forms

There is an important feature of the symfony form framework we haven't considered yet: embedded forms. If an instance of `CommentForm` is embedded in another form, the enhancements we've made in `forms.yml` will not be applied. This is easy enough to demonstrate in our test script:

Listing 14-19

```
$t = new lime_test(6);

// ...

$form = new BaseForm();
$form->embedForm('comment', new CommentForm());
$form->bind();
$enhancer->enhance($form);
$t->like($form['comment']['body']->renderLabel(),
  '/Please enter your comment/',
  '->enhance() enhances embedded forms');
```

This new assertion demonstrates that embedded forms are not being enhanced:




Fixing this test will involve a more advanced config handler. We need to be able to apply the enhancements configured in `forms.yml` in a modular way to account for embedded forms, so we are going to generate a tailored enhancer method for each configured form class. These methods will be generated by our custom config handler in a new "worker" class.

Listing 14-20

```
class sfFormYamlEnhancementsConfigHandler extends sfYamlConfigHandler
{
  // ...

  protected function getEnhancerCode($fields)
  {
    $code = array();
    foreach ($fields as $field => $config)
    {
      $code[] = sprintf('if (isset($fields[%s]))', var_export($field,
true));
      $code[] = '{';
```

```
      if (isset($config['label']))
      {
        $code[] = sprintf('  $fields[%s]->getWidget()->setLabel(%s);',
          var_export($config['label'], true));
      }

      if (isset($config['attributes']))
      {
        $code[] = '  $fields[%s]->getWidget()->setAttributes(array_merge(';
        $code[] = '    $fields[%s]->getWidget()->getAttributes(),';
        $code[] = '    '.var_export($config['attributes'], true);
        $code[] = '  ));';
      }

      if (isset($config['errors']))
      {
        $code[] = sprintf('  if ($error = $fields[%s]->getError())',
          var_export($field, true));
        $code[] = '  {';
        $code[] = '    $error->getValidator()->setMessages(array_merge(';
        $code[] = '      $error->getValidator()->getMessages(),';
        $code[] = '      '.var_export($config['errors'], true);
        $code[] = '    ));';
        $code[] = '  }';
      }

      $code[] = '}';
    }

    return implode(PHP_EOL.'    ', $code);
  }
}
```

Notice how the config array is checked for certain keys when the code is generated, rather than at runtime. This will provide a small performance boost.

As a general rule, logic that checks conditions of the configuration should be run in the config handler, not in the generated code. Logic that checks runtime conditions, such as the nature of the form object being enhanced, must be run at runtime.

This generated code is placed inside a class definition, which is then saved to the cache directory.

*Listing 14-21*

```
class sfFormYamlEnhancementsConfigHandler extends sfYamlConfigHandler
{
  public function execute($configFiles)
  {
    $forms = self::getConfiguration($configFiles);

    $code = array();
    $code[] = '<?php';
    $code[] = '// auto-generated by '.__CLASS__;
    $code[] = '// date: '.date('Y/m/d H:is');
    $code[] = 'class sfFormYamlEnhancementsWorker';
    $code[] = '{';
    $code[] = '  static public $enhancable =
'.var_export(array_keys($forms), true).';';
```

```
    foreach ($forms as $class => $fields)
    {
      $code[] = '  static public function
enhance'.$class.'(sfFormFieldSchema $fields)';
      $code[] = '  {';
      $code[] = '    '.$this->getEnhancerCode($fields);
      $code[] = '  }';
    }

    $code[] = '}';

    return implode(PHP_EOL, $code);
  }

  // ...
}
```

The `sfFormYamlEnhancer` class will now defer to the generated worker class to handle manipulation of form objects, but must now account for recursion through embedded forms. To do this we must process the form's field schema (which can be iterated through recursively) and the form object (which includes the embedded forms) in parallel.

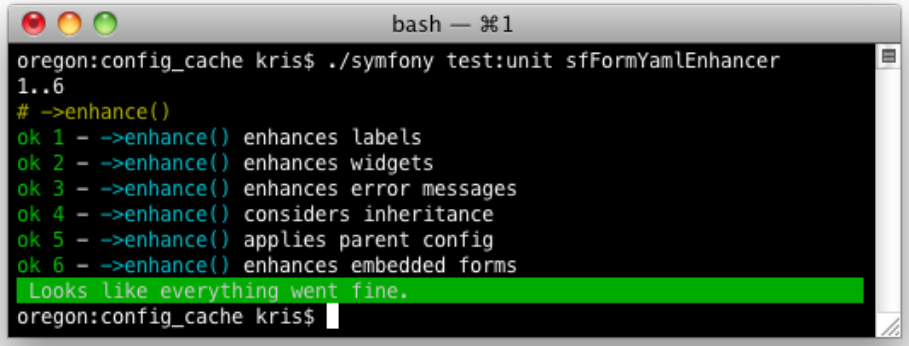
*Listing 14-22*

```
class sfFormYamlEnhancer
{
  // ...

  public function enhance(sfForm $form)
  {
    require_once $this->configCache->checkConfig('config/forms.yml');
    $this->doEnhance($form->getFormFieldSchema(), $form);
  }

  protected function doEnhance(sfFormFieldSchema $fieldSchema, sfForm
$form)
  {
    if ($enhancer = $this->getEnhancer(get_class($form)))
    {
      call_user_func($enhancer, $fieldSchema);
    }

    foreach ($form->getEmbeddedForms() as $name => $form)
    {
      if (isset($fieldSchema[$name]))
      {
        $this->doEnhance($fieldSchema[$name], $form);
      }
    }
  }

  public function getEnhancer($class)
  {
    if (in_array($class, sfFormYamlEnhancementsWorker::$enhancable))
    {
      return array('sfFormYamlEnhancementsWorker', 'enhance'.$class);
    }
    else if ($overlap = array_intersect(class_parents($class),
      sfFormYamlEnhancementsWorker::$enhancable))
```

```
    {
      return array('sfFormYamlEnhancementsWorker',
'enhance'.current($overlap));
    }
  }
}
```

The fields on embedded form objects should not be modified after they've been embedded. Embedded forms are stored in the parent form for processing purposes, but have no effect on how the parent form is rendered.

With support for embedded forms in place, our tests should now be passing. Run the script to find out:




## How'd we do?

Let's run some benchmarks just to be sure we haven't wasted our time. To make the results interesting, add a few more form classes to `forms.yml` using a PHP loop.

*Listing 14-23*
```
# <?php for ($i = 0; $i < 100; $i++): ?> #
Form<?php echo $i ?>: ~
# <?php endfor; ?> #
```

Create all these classes by running the following snippet of code:

*Listing 14-24*
```
mkdir($dir = sfConfig::get('sf_lib_dir').'/form/test_fixtures');
for ($i = 0; $i < 100; $i++)
{
  file_put_contents($dir.'/Form'.$i.'.class.php',
    '<?php class Form'.$i.' extends BaseForm { }');
}
```

Now we're ready to run some benchmarks. For the results below, I've ran the following Apache[144] command on my MacBook multiple times until I got a standard deviation of less than 2ms.

*Listing 14-25*
```
$ ab -t 60 -n 20 http://localhost/config_cache/web/index.php
```

144. `http://httpd.apache.org/docs/2.0/programs/ab.html`

Start with a baseline benchmark for running the application without the enhancer connected at all. Comment out `sfFormYamlEnhancer` in `frontendConfiguration` and run the benchmark:

Listing 14-26

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:    62   63   1.5     63      69
Waiting:       62   63   1.5     63      69
Total:         62   63   1.5     63      69
```

Next, paste the first version of `sfFormYamlEnhancer::enhance()` that called `sfYaml` directly into the class and run the benchmark:

Listing 14-27

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:    87   88   1.6     88      93
Waiting:       87   88   1.6     88      93
Total:         87   88   1.7     88      94
```

You can see we've added an average of 25ms to each request, an increase of almost 40%. Next, undo the change you just made to `->enhance()` so our custom config handler is restored and run the benchmark again:

Listing 14-28

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:    62   63   1.6     63      70
Waiting:       62   63   1.6     63      70
Total:         62   64   1.6     63      70
```

As you can see, we've reduced processing time back to the baseline by creating a custom config handler.

# Just For Fun: Bundling a Plugin

Now that we have this great system in place for enhancing form objects with a simple YAML configuration file, why not bundle it up as a plugin and share it with the community. This may sound intimidating to those who haven't published a plugin in the past; hopefully we can dispell some of that fear now.

This plugin will have the following file structure:

Listing 14-29

```
sfFormYamlEnhancementsPlugin/
  config/
    sfFormYamlEnhancementsPluginConfiguration.class.php
  lib/
    config/
      sfFormYamlEnhancementsConfigHandler.class.php
    form/
      sfFormYamlEnhancer.class.php
  test/
    unit/
      form/
        sfFormYamlEnhancerTest.php
```

We need to make a few modifications to ease the plugin installation process. Creation and connection of the enhancer object can be encapsulated in the plugin configuration class:

Listing 14-30
```
class sfFormYamlEnhancementsPluginConfiguration extends
sfPluginConfiguration
{
  public function initialize()
  {
    if ($this->configuration instanceof sfApplicationConfiguration)
    {
      $enhancer = new
sfFormYamlEnhancer($this->configuration->getConfigCache());
      $enhancer->connect($this->dispatcher);
    }
  }
}
```

The test script will need to be updated to reference the project's bootstrap script:

Listing 14-31
```
include dirname(__FILE__).'/../../../../../test/bootstrap/unit.php';

// ...
```

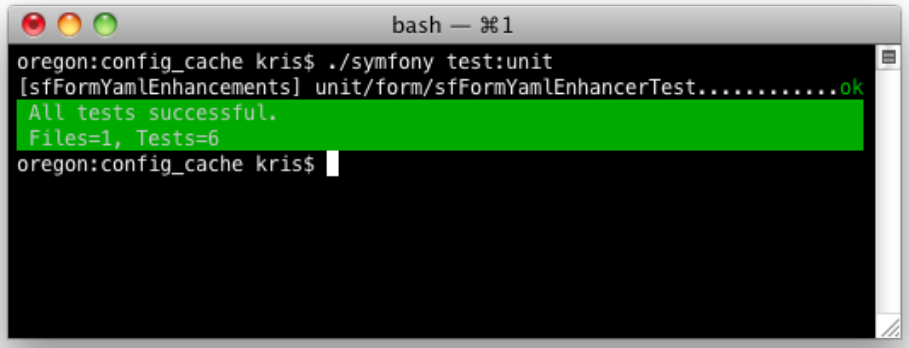
Finally, enable the plugin in `ProjectConfiguration`:

Listing 14-32
```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->enablePlugins('sfFormYamlEnhancementsPlugin');
  }
}
```

If you want to run tests from the plugin, connect them in `ProjectConfiguration` now:

Listing 14-33
```
class ProjectConfiguration extends sfProjectConfiguration
{
  // ...

  public function setupPlugins()
  {

$this->pluginConfigurations['sfFormYamlEnhancementsPlugin']->connectTests();
  }
}
```

Now the tests from the plugin will run when you call any of the `test:*` tasks.




All of our classes are now located in the new plugin's directory, but there is one problem: the test script relies on files that are still located in the project. This means that anyone else who may want to run these tests would not be able to unless they have the same files in their project.

To fix this we'll need to isolate the code in the enhancer class that calls the config cache so we can overload it in our test script and instead use a `forms.yml` fixture.

*Listing 14-34*

```
class sfFormYamlEnhancer
{
  // ...

  public function enhance(sfForm $form)
  {
    $this->loadWorker();
    $this->doEnhance($form->getFormFieldSchema(), $form);
  }

  public function loadWorker()
  {
    require_once $this->configCache->checkConfig('config/forms.yml');
  }

  // ...
}
```

We can then overload the `->loadWorker()` method in our test script to call the custom config handler directly. The `CommentForm` class should also be moved to the test script and the `forms.yml` file moved to the plugin's `test/fixtures` directory.

*Listing 14-35*

```
include dirname(__FILE__).'/../../../../../test/bootstrap/unit.php';

$t = new lime_test(6);

class sfFormYamlEnhancerTest extends sfFormYamlEnhancer
{
  public function loadWorker()
  {
    if (!class_exists('sfFormYamlEnhancementsWorker', false))
    {
      $configHandler = new sfFormYamlEnhancementsConfigHandler();
      $code = $configHandler->execute(array(dirname(__FILE__).'/../../
fixtures/forms.yml'));
```

```
      $file = tempnam(sys_get_temp_dir(), 'sfFormYamlEnhancementsWorker');
      file_put_contents($file, $code);

      require $file;
    }
  }
}

class CommentForm extends BaseForm
{
  public function configure()
  {
    $this->setWidget('body', new sfWidgetFormTextarea());
    $this->setValidator('body', new sfValidatorString(array('min_length'
=> 12)));
  }
}

$configuration = $configuration->getApplicationConfiguration(
  'frontend', 'test', true, null, $configuration->getEventDispatcher());

$enhancer = new sfFormYamlEnhancerTest($configuration->getConfigCache());

// ...
```

Finally, packaging the plugin is easy with `sfTaskExtraPlugin` installed. Just run the `plugin:package` task and a package will be created after a few interactive prompts.

*Listing 14-36*

```
$ php symfony plugin:package sfFormYamlEnhancementsPlugin
```

The code in this article has been published as a plugin and is available to download from the symfony plugins site:

http://symfony-project.org/plugins/sfFormYamlEnhancementsPlugin

This plugin includes what we've covered here and much more, including support for `widgets.yml` and `validators.yml` files as well as integration with the `i18n:extract` task for easy internationalization of your forms.

# Final Thoughts

As you can see by the benchmarks done here, the symfony config cache makes it possible to utilize the simplicity of YAML configuration files with virtually no impact on performance.

Chapter 15

# Working with the symfony Community

*by Stefan Koopmanschap*

There are many reasons to work with Open-Source software. Being able to see the source code is one. The software is often free. But one of the most important reasons to choose Open-Source is the community. There's all kinds of communities around Open-Source projects depending on the project. The symfony community is generally described as being open and friendly. But how can you get the most from your relationship with the community? And what are the best ways to contribute something back? In this chapter I want to introduce you to the symfony community and the ways to work with it. Both individual people and companies will be able to get some inspiration for the best way to interact with the community and get the most out of it.

## Getting the best from the Community

There's many ways to get something out of the symfony community. Some ways are so integrated into the usage of symfony as a framework that you might not even consider that it is actually only due to the community that you can do that. The main thing, of course, is using symfony itself. Even though symfony was initially developed by a company and still has a strong backing by Sensio, the framework would never be where it is today without a strong community backing it as well. So let's see how you gain from the community, besides the framework itself.

### Support

Every developer, especially developers just starting out with the framework, will get to a point where they have no idea how to proceed. You're stuck at a point where you're just puzzled on the best way forward. Luckily, symfony has a very friendly community that will help you out with just about any question you might have. There are various ways to get your answers, depending on your preference. The basic concept is usually the same though: You ask a question and most of the time, you get a speedy answer.

#### Before asking a Question

Before you ask a question using one of the below methods, it is best to first look for the solution yourself. You can of course use Google[145] to search the wider web (and you should!)

145. `http://www.google.com/`

but if you want to search a bit more specific, the archives of the different mailinglists, especially the symfony-users archives[146], are a good place to start.

### Asking a Question

It is important to know how to ask a question. It may sound simple but it is actually important to first think about what you're asking. Also, make sure you have checked the existing documentation to see if your question has already been answered. There are some general considerations that will help you get a better response:

- Think about your question. Make sure you formulate your question clearly. State what you are doing (or trying to do), what you are unable to do, and make sure to clearly mention any errors that you are getting.
- Offer a short history of things you've tried. Reference the documentation you used to try to solve the problem, the possible solutions you found while searching the web or the mailinglist archives, or any of the things you tried while trying to solve the problem

### Mailinglists

There are several Google Groups[147] around symfony for different purposes. These mailinglists are the premier way of getting in touch with users and developers of symfony. If you are working with symfony and looking for support on a problem you have, then the symfony users[148] is the place to be. The subscribers to this mailinglist are a mixture of regular symfony users, symfony beginners and most of the symfony core team. For any given question there is usually someone that is able to answer it. There are several other lists for other purposes:

- symfony-devs[149]: For discussions on symfony core development (*not for support!*)
- symfony-docs[150]: For discussions on symfony documentation
- symfony-community[151]: For discussions on community initiatives

With any of these mailinglists, you should keep in mind that they are less direct in their nature than other means of communication such as IRC. It can take hours or even up to a few days to get the response you are looking for. It is important though to be responsive on any additional questions people might have and not be too impatient.

Opposite to IRC, you should give as much background information with your question. Information on configuration, which ORM you use, what kind of operating system you are using, which possible solutions you tried and what failed. Any code samples can be included in the e-mail, to give your problem as much context as possible which may lead to the solution.

### IRC

IRC is the most direct way to get answers because of its real-time nature. Symfony has a dedicated channel called #symfony on the Freenode network[152] where on any given time during the day many people hang out. Be aware though that even though the channel may have over 100 users present, often many of these people are at work and don't constantly

146. `http://groups.google.com/group/symfony-users/topics`
147. `http://groups.google.com`
148. `http://groups.google.com/group/symfony-users`
149. `http://groups.google.com/group/symfony-devs`
150. `http://groups.google.com/group/symfony-docs`
151. `http://groups.google.com/group/symfony-community`
152. `http://freenode.net/`

check the IRC channel. So, though IRC is real-time in nature, it can sometimes take a while to get a response.

The nature of IRC does not really allow for the displaying of big blocks of code and such. So describe your problem in the IRC channel, but if you want to show blocks of code or contents of configuration files, be sure to use sites like pastebin.com and only reference the link in the IRC channel. People often see the pasting of blocks of code as flooding the channel. This is not appreciated and you might have a harder time getting an answer.

When asking a question in IRC, pay attention to the response you get. Be responsive to any additional questions you may get about your problem. Be aware that sometimes people in the IRC channel may even question your whole approach to the problem. Sometimes they may be right, sometimes they may judge your situation incorrectly, but make sure that you answer their questions and describe your situation when people ask for it. They won't know your whole project layout and so may make assumptions that prove incorrect once you explain it more clearly. Don't feel offended by such things: The people in the channel are only trying to help you.

When the channel is a bit busy, make sure you prefix any answers to specific questions with the name of the person you are answering a question to. This will make it more clear who you are talking to for those asking the question but also for those in other conversations; they now know that they can ignore your message because it is not a part of their conversation.

## Fixes and new Features

This is something we all seem to take for granted, but needs to be said: The whole symfony codebase is one big community effort. There's a lot of time in there from Sensio and Fabien specifically, but even their work is community work because, by releasing it as Open-Source, they've shown their heart for the community. But all the other developers that also work on either writing new functionality of supplying bugfixes are doing this for the community. So when you are working with symfony (or any other Open-Source project for that matter) be aware that it is thanks to the community that you are able to use the software.

## Plugins

Symfony has a very extensive plugin system, which allows you to easily install external plugins into your symfony project. The plugin system is based on the PEAR installer and channel system, and is very flexible. Aside from the plugins that are included in the symfony distribution, there is a huge amount of plugins that have been developed and are maintained by the community. You can go to the plugin site[153] and browse through the plugins based on categories, ORM used, symfony version that is supported and also search. Thanks to the work of the community you can find plugins for a lot of the common functionality you find in today's web applications.

## Conferences and Events

Aside from interacting with the community through digital means as described before, you can also communicate with the community at conferences and other events. Most of the PHP-related conferences and even some conferences aimed at a wider audience than just PHP have some symfony community members attending or even presenting. It is the work of these community members that will help you learn from your peers at these conferences. There are events that are completely built up around symfony. Some great examples of those are Symfony Live[154], Symfony Day[155] and SymfonyCamp[156]. All these events have the backing of

153. http://www.symfony-project.org/plugins/
154. http://www.symfony-live.com/
155. http://www.symfonyday.com/

a company but most of the work is a community effort, and by attending such an event you can learn a lot about symfony and get in touch with those key community members that can help you a bit further when you're stuck somewhere. If you have the opportunity of attending such an event, it is definitely recommended to do so.

Aside from the above conferences, there is also a growing amount of symfony user groups around the world. These user groups usually don't have any company backing, but are just a gathering of symfony users on a regular basis in a specific area. It is very easy to go to one of these gatherings, the only effort it takes is to show up at one of the meetings. These meetings will allow you to build up a network of symfony-related people who can help you with symfony problems, might have a job for you if you're looking for one, or know a developer that is available when you need one.

### Reputation

Being present in the community, being seen by people, talking to people, perhaps also becoming more active in the community, will allow you to build a reputation. At first, this may seem useless except that it is a nice ego boost, but you can gain much more from that reputation. When you're looking for a job, and notify the community of the fact, often you will get contacted by one or two people to see if you fit their job opening. But once you've started building a bit of a reputation, the amount of offers you get may grow and the positions may also become more interesting.

Similarly, when you are looking for developers and notify the community about it, you may get a few responses. As your reputation in the community grows, the amount of responses may grow and you may even be able to pick from some of the bigger names in the community for your job opening.

## Giving back to the Community

Any community works only as a give-and-take system. If there was no one to offer something to the community, there would be nothing to get from the community either, and the community would not exist. So as you get something from the community, it might also be good to think about what you can do back. How can you help the community be a better community and grow? How can you contribute to the existence and strength of the community? Let's go through some ways you might be able to contribute.

### The Forums and Mailinglists

As described earlier, the forums and mailinglists are a place where you can get support. You can get answers to your questions, suggestions on how to solve problems, or feedback on a specific approach you've taken for your project. And even though you might have just started out with symfony, as you're getting some experience with symfony, you might be able to answer the basic questions of other starting symfony users. And as you grow more experienced with symfony, you'll be able to answer more advanced questions and join in discussions of such issues. Even simply offering a direction a user may want to look into can help people find the solution to their problem. Since you might already be subscribed to a mailinglist, adding your expertise is a small amount of effort that will help others.

### IRC

More direct communication around symfony is done on IRC, as described before. Just as with the mailinglists, if you're hanging out in the IRC channel anyway, you can scan it every once

156. `http://www.symfonycamp.com/`

in a while for questions that you might be able to answer. There is no need to keep a constant watch on the IRC channel, most people in the #symfony channel don't keep a close watch all the time. Instead, when they need a short break from their work, they switch to their IRC client, check what discussions are going on, and try to give their input in an effort to clarify any issues, help people or simply discuss a specific feature. Being present in IRC, even though you don't directly check the channel every minute, also allows people to "highlight" you by mentioning your nickname. Most IRC clients will then notify you of the highlight, and you can then respond. This will make you more approachable by the community if they have a question that they know you might be able to answer. So even when doing nothing, you are doing something useful: Being available.

## Contributing Code

Probably the easiest way for people who work with symfony to contribute is to actually contribute some code. Since symfony users are all developers, this is the most fun way to give something back to the community. There are several ways of contributing code. Below if a list of ways to do that.

### Core Patches

It can, of course, happen that while working with symfony you run into a bug. Or, perhaps you want to do something, but find out symfony does not offer the specific feature you would like to use. Working with a patched version of a framework is not recommended since it might be a problem as soon as you update the version. Also, you might forget about the patch and run into problems later. And, especially when you run into a bug, it is good practice to notify the framework developers of the issue. So, how can you actually do this?

First of all, the changes need to be made to the symfony code. The framework files should be altered to fix the bug or add the behaviour. Then, assuming the changes are done in a Subversion checkout of the symfony code, a diff can be made of the changes done to the symfony codebase. With Subversion, this could be done by issuing the following command:

```
$ svn diff > my_feature_or_bug_fix.patch
```

*Listing 15-1*

This command should be issued in the root of the symfony checkout, to ensure all changes that were made in the symfony code end up in the patch file.

The next step would be to go to the symfony bugtracker[157]. After logging in, a new option appears to create a new ticket. When creating a new ticket, make sure to fill in as many fields as possible, to make it easier for the core team to reproduce bugs and/or know what parts of symfony are affected.

In the "Ticket Properties" box, make sure to select the right version of symfony the patch was based on. Additionally, when possible, select the component of symfony that is affected by the patch. When there are multiple components affected, select the one that is most affected and make sure to mention in the "Full Description" field which parts of symfony are affected.

Important to note is that the "Short Summary" field should contain the prefix [PATCH] when a patch is attached. At the bottom of the form, make sure to check the checkbox to indicate there is a patch file to attach to the new ticket.

### Contributing Plugins

Hacking into the framework core is not for everyone. But symfony users work on symfony projects that contain custom functionality. Some functionality is very specific to a project so won't be very useful to open up for others to use, but often a project contains very generic

157. http://trac.symfony-project.org

code that could be of use to others. It is a best practice to put a lot of the application logic in plugins to be able to reuse them easily, at the very least, internally in the organization. But, given the code was put into a plugin, it is also possible to Open-Source it and make it available for all symfony users.

Contributing a plugin to the symfony community is quite easy. You can read the documentation[158] about how to create and package the plugin and upload it to the symfony website. The symfony site allows plugin developers to use a full set of tools to publish through the symfony plugin channel server and also to host the plugin source in a Subversion repository on the symfony server so that other developers can access the code. When considering opening up a plugin, this is the recommended way of publishing the code. This is much easier than managing a Subversion server and PEAR package server and creating the documentation to explain to users how to use the custom systems. Adding a plugin to the symfony plugin system will automatically make it available to all symfony users without additional configuration. Obviously though, it is still possible to create a PEAR package server and have users add that to their projects to install plugins.

## Documentation

One of the very strong parts of symfony is its documentation. The core team has written a lot of documentation on how to use symfony, but a big part of the documentation is also there thanks to the community. There are also joint efforts of the core team and the community, such as the work on the Jobeet tutorial. Documentation helps new people learn symfony and also helps as a reference for experienced developers, so it is very important to have good documentation. There are several ways of contributing documentation for symfony.

### Write on your Weblog

Sharing experiences and knowledge on symfony is very important for the community. Especially when running into something that is hard to figure out, sharing it with the community is a good thing to do. Other people may have the same problems and might be using a search engine to find out if other people have had similar issues. Having some good search results will help solve their issue much faster.

So when writing a blog post, the topic does not need to be a generic introduction into symfony (of course it can be!), but can be about experiences with the framework, solutions to problems that were encountered while working with the framework, or a cool new feature of the latest version.

Anyone writing about symfony can add their weblog to the symfony bloggers list. All weblogs on this list are syndicated on the symfony community page[159]. There are some guidelines however: A symfony-specific feed is requested so that all content on the community page is symfony related. Also, please don’t add anything other than weblogs (so no twitter feeds for instance).

### Writing Articles

People that are more comfortable with their writing can take writing one step further. There are several PHP magazines around the world as well as many computer magazines that allow people to propose articles. Articles for such publications are usually more advanced, more structured and of higher quality than the average blogpost, but are also read by many more people.

158. `http://www.symfony-project.org/jobeet/1_3/Doctrine/en/20#chapter_20_contributing_a_plugin`
159. `http://www.symfony-project.org/community`

Most publications have their own way of accepting article submissions, so check with the magazine website or in the printed magazine to see what steps to take to submit content to the magazine.

Aside from magazines, there are other places where articles are usually welcomed. For instance, websites of PHP usergroups or symfony usergroups, generic web development websites, and other online publications often appreciate articles with good content that can be published on their website.

### Translating Documentation

Most people doing PHP development these days will be familiar with the English language. However, for many people it is not their native tongue making it hard for them to read extensive technical content. Symfony promotes the translation of documentation and supports it by offering write access to the documentation repository for translators and publishing translated versions of the documentation on the symfony website.

Translations of the documentation is mostly coordinated and discussed through the symfony docs mailinglist[160]. If you are interested in helping with translating the documentation to your native language, this would be the first place to check. There is a chance that there are multiple translators for a language, in which case it is very important to coordinate efforts so as to not duplicate any work. The symfony docs mailinglist is the perfect place to start your translations efforts.

### Add Content to the Wiki

A wiki is one of the most open ways of documentation on any topic. Symfony has a wiki[161] where people can add documentation. It is always good to get more new content on the wiki. However, it is also possible to contribute by looking at the existing articles on the wiki and correcting and/or updating them. Aside from new articles, there are also old articles that have outdated examples or may even be fully outdated. Helping to clean up the existing content on the wiki is a great way to make it easier for people searching through the wiki to find the right content.

If you want to get an idea of what kind of content is on the wiki or get some inspiration for content you want to write, just have a look at the wiki homepage[162] to see what is already there.

## Presentations

Writing is a good way of sharing knowledge and experiences. The content is available to a lot of people and is searchable. However, there are more ways to get your experience and knowledge across. One good way that many people appreciate is by doing presentations. You can do presentations in many different settings and for many different audiences. For instance:

- At PHP/symfony conferences
- At local (PHP) user group meetings
- Inside your company (for your developer colleagues)
- Inside your company (for management)

Depending on the location and also your target audience, you will have to adapt your presentation. While management will not be interested in all the technical details, the audience at a symfony conference will not need a basic introduction to symfony. Take your

160. `http://groups.google.com/group/symfony-docs`
161. `http://trac.symfony-project.org/wiki`
162. `http://trac.symfony-project.org/wiki`

time to pick the right topic and prepare a presentation. Have the slides reviewed by someone else and, if possible, do a try-out of the presentation for some people who can and will give honest feedback: not just praise, but also criticism so you can improve your presentation before you do it for real.

Additional help on preparing and doing presentations is always available on the symfony community mailinglist[163], where experienced speakers as well as regular conference visitors will be able to help with tips, tricks and experiences. Also, if you don't know of a conference or user group where you could do your presentation, you can subscribe to the mailinglist to get updates on Call for Papers of conferences or contacts with usergroups.

## Organize an Event/Meetup

Aside from doing presentations on existing conferences and meetings, you can also organize something yourself. It can be very small or very big. It can be aimed at the worldwide community, or just your local users. It can even be part of an existing event.

One example of this was the ad-hoc symfony update meeting that was held at the PHPNW conference in 2008. This all started on the Twitter and IRC backchannel for the conference, where several symfony users had some questions on what symfony 1.2 would be like. Eventually a room was arranged thanks to the organization and during one of the breaks between sessions, a group of approximately 10 people gathered to get an update of what symfony 1.2 would all be about. It was small and simple yet very effective as those present got an idea of what to expect in the (at that time) new version of symfony.

Another great example is the organization of community conferences such as SymfonyCamp[164] and SymfonyDay Cologne[165]. Both symfony conferences were organized by PHP development companies working with symfony and trying to give something back to the community. All of these conferences were well-visited, had a great speaker schedule and had a very nice community feel to them.

## Become locally active

It was mentioned before that not everyone is able to understand a lot of technical content in english. Also, it can sometimes be nice to not only communicate online about symfony. You can also become locally active. A good example for this is local symfony user groups. Over the past year, there have been several initiatives for the starting of new user groups, and most have by now organized several get-togethers for people interested in symfony or working with the framework. Most of these events are very informal, free, and are completely community-driven.

The earlier mentioned symfony community mailinglist[166] is a great place to look for an existing usergroup in your area, as well as the place to initiate a new symfony usergroup. There are members and organizers of local symfony usergroups on the list that can offer their help with starting a new local usergroup and getting organized.

Aside from the physical activities you can organize locally, you can also try to promote symfony in your region on-line. One good example is by starting a local symfony portal. A good example of such a local portal is the Spanish symfony portal[167], which has regular updates on what is happening with symfony posted to the website in Spanish. The site also provides extensive Spanish documentation, so it is a great way for Spanish developers to learn symfony and keep up-to-date with the new developments around symfony.

163. http://groups.google.com/group/symfony-community
164. http://www.symfonycamp.com/
165. http://www.symfonyday.com/
166. http://groups.google.com/group/symfony-community
167. http://www.symfony.es/

## Becoming part of the Core Team

The core team is of course also a part of the community. The people that are in the core team all started out as users of the framework and due to their involvement in one way or another became a part of the core team. Symfony is a meritocracy[168], which means that if you prove your skills and/or talent, you might be able to become part of the core team as well.

A good example of this is the joining of Bernhard Schussek[169]. Bernhard joined the core team after his fantastic work on the second version of the Lime testing framework and having submitted patches consistently for a long period of time.

## Where to start?

Now that you've heard what you can get from the community and how you can contribute, it might be nice to have a short overview of the starting points for getting involved in the symfony community. Use these to find your way around the community.

### Symfony-community Mailinglist

The symfony-community mailinglist[170] is a mailinglist where members can discuss ideas for community efforts, join community efforts currently being executed, and throw in anything community-related. If you want to join one of these efforts, simply reply to the discussion about the effort. If you have any new ideas that might be useful for the symfony community, you can post them on this list. If you have questions about the community or the different ways of communicating with the community, this is also the place to be.

### The "How to contribute to symfony" page

For quite some time now, symfony has had a special page on the wiki titled How to contribute to symfony[171]. This page lists all the ways you can get involved in helping the symfony project and community with whatever skills you have, and links to the right places for most of these ways to help. It is a recommended read for anyone who wants to get involved in the symfony community.

## External Communities

Please don't stop at becoming involved in the symfony community as described in this article. There's a lot of initiatives worldwide that are started around symfony and users of the framework. I want to give some extra attention to two of these as they may be very useful for people working with symfony.

### Symfonians

The Symfonians community[172] is a community which lists people and companies that work with symfony and projects done with symfony. The site also allows companies to post job openings and allows you to search within your own country for people, companies and projects.

Aside from the site being a great way to get in touch with other symfony users or even find a job, the application directory is a great showcase of what you can do with symfony. There is

168. http://en.wikipedia.org/wiki/Meritocracy
169. http://www.symfony-project.org/blog/2009/08/27/bernhard-schussek-joins-the-core-team
170. http://groups.google.com/group/symfony-community
171. http://trac.symfony-project.org/wiki/HowToContributeToSymfony
172. http://www.symfonians.net/

an enormous variety in the nature of the applications listed on the site, making it very nice to browse through to see what people do with the framework.

Since all content on the site is contributed by the community, you can also get an account and create your own profile, your company profile, add applications you've built with symfony or post job openings.

### LinkedIn symfony Group

Any professional PHP developer has most probably encountered LinkedIn. Most will have their own LinkedIn profile. For those who don't know LinkedIn: It is a network site where you can maintain your own network and keep in touch with the people in your network.

LinkedIn also offers a groups feature, allowing group discussions, news and job postings. Many topics have a LinkedIn group, and symfony is no exception[173] (login required). Using this symfony group, you can discuss symfony-related topics, follow symfony-related news and also post jobs you have for symfony developers, trainers, consultants and architects.

## Final Thoughts

By now, you should have a good idea on what you can expect from the community, and what the community may expect from you. Remember that any open source software relies on a community being able to support the software. But this support can be anything, from answering some questions to submitting patches and plugins to promote the software. It would be great if you would join us!

173. `http://www.linkedin.com/groups?gid=29205&trk=myg_ugrp_ovr`

# Appendices

Appendix A

# JavaScript code for sfWidgetFormGMapAddress

The following code is the JavaScript needed to make the `sfWidgetFormGMapAddress` widget work:

*Listing A-1*

```
function sfGmapWidgetWidget(options){
  // this global attributes
  this.lng      = null;
  this.lat      = null;
  this.address  = null;
  this.map      = null;
  this.geocoder = null;
  this.options  = options;

  this.init();
}

sfGmapWidgetWidget.prototype = new Object();

sfGmapWidgetWidget.prototype.init = function() {

  if(!GBrowserIsCompatible())
  {
    return;
  }

  // retrieve dom element
  this.lng      = jQuery("#" + this.options.longitude);
  this.lat      = jQuery("#" + this.options.latitude);
  this.address  = jQuery("#" + this.options.address);
  this.lookup   = jQuery("#" + this.options.lookup);

  // create the google geocoder object
  this.geocoder = new GClientGeocoder();

  // create the map
  this.map = new GMap2(jQuery("#" + this.options.map).get(0));
  this.map.setCenter(new GLatLng(this.lat.val(), this.lng.val()), 13);
  this.map.setUIToDefault();

  // cross reference object
  this.map.sfGmapWidgetWidget = this;
```

```
  this.geocoder.sfGmapWidgetWidget = this;
  this.lookup.get(0).sfGmapWidgetWidget = this;

  // add the default location
  var point = new GLatLng(this.lat.val(), this.lng.val());
  var marker = new GMarker(point);
  this.map.setCenter(point, 15);
  this.map.addOverlay(marker);

  // bind the move action on the map
  GEvent.addListener(this.map, "move", function() {
     var center = this.getCenter();
     this.sfGmapWidgetWidget.lng.val(center.lng());
     this.sfGmapWidgetWidget.lat.val(center.lat());
  });

  // bind the click action on the map
  GEvent.addListener(this.map, "click", function(overlay, latlng) {
    if (latlng != null) {
      sfGmapWidgetWidget.activeWidget = this.sfGmapWidgetWidget;

      this.sfGmapWidgetWidget.geocoder.getLocations(
        latlng,
        sfGmapWidgetWidget.reverseLookupCallback
      );
    }
  });

  // bind the click action on the lookup field
  this.lookup.bind('click', function(){
    sfGmapWidgetWidget.activeWidget = this.sfGmapWidgetWidget;

    this.sfGmapWidgetWidget.geocoder.getLatLng(
      this.sfGmapWidgetWidget.address.val(),
      sfGmapWidgetWidget.lookupCallback
    );

    return false;
  })
}

sfGmapWidgetWidget.activeWidget = null;
sfGmapWidgetWidget.lookupCallback = function(point)
{
  // get the widget and clear the state variable
  var widget = sfGmapWidgetWidget.activeWidget;
  sfGmapWidgetWidget.activeWidget = null;

  if (!point) {
    alert("address not found");
    return;
  }

  widget.map.clearOverlays();
  widget.map.setCenter(point, 15);
  var marker = new GMarker(point);
  widget.map.addOverlay(marker);
}
```

```
sfGmapWidgetWidget.reverseLookupCallback = function(response)
{
  // get the widget and clear the state variable
  var widget = sfGmapWidgetWidget.activeWidget;
  sfGmapWidgetWidget.activeWidget = null;

  widget.map.clearOverlays();

  if (!response || response.Status.code != 200) {
    alert('no address found');
    return;
  }

  // get information location and init variables
  var place = response.Placemark[0];
  var point = new
GLatLng(place.Point.coordinates[1],place.Point.coordinates[0]);
  var marker = new GMarker(point);

  // add marker and center the map
  widget.map.setCenter(point, 15);
  widget.map.addOverlay(marker);

  // update values
  widget.address.val(place.address);
  widget.lat.val(place.Point.coordinates[1]);
  widget.lng.val(place.Point.coordinates[0]);
}
```

Appendix B

# Custom Installer Example

The following PHP code is a custom installer used in Chapter 06 *(page 66)*:

Listing B-1

```
<?php

$this->logSection('install', 'default to sqlite');
$this->runTask('configure:database', sprintf("'sqlite:%s/database.db'",
sfConfig::get('sf_data_dir')));

$this->logSection('install', 'create an application');
$this->runTask('generate:app', 'frontend');

$this->setConfiguration($this->createConfiguration('frontend', 'dev'));

$this->logSection('install', 'publish assets');
$this->runTask('plugin:publish-assets');
if (file_exists($dir = sfConfig::get('sf_symfony_lib_dir').'/../data'))
{
  $this->installDir($dir);
}

$this->logSection('install', 'create the database schema');
file_put_contents(sfConfig::get('sf_config_dir').'/doctrine/schema.yml',
<<<EOF
Product:
  columns:
    name:           { type: string(255), notnull: true }
    price:          { type: decimal, notnull: true }

ProductPhoto:
  columns:
    product_id:     { type: integer }
    filename:       { type: string(255) }
    caption:        { type: string(255), notnull: true }
  relations:
    Product:
      alias:        Product
      foreignType:  many
      foreignAlias: Photos
      onDelete:     cascade
EOF
);

$this->logSection('install', 'add some fixtures');
```

```
file_put_contents(sfConfig::get('sf_data_dir').'/fixtures/fixtures.yml',
<<<EOF
Product:
  product_1:
    name:  Product Name
    price: 25.95
EOF
);

$this->logSection('install', 'build the model');
$this->runTask('doctrine:build', '--all --and-load --no-confirmation');

$this->logSection('install', 'create a simple CRUD module');
$this->runTask('doctrine:generate-module', 'frontend product Product
--non-verbose-templates');

$this->logSection('install', 'fix sqlite database permissions');
chmod(sfConfig::get('sf_data_dir'), 0777);
chmod(sfConfig::get('sf_data_dir').'/database.db', 0777);
```

Appendix C

# License

## Attribution-Share Alike 3.0 Unported License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

   b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

   c. **"Creative Commons Compatible License"** means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this

License or a Creative Commons jurisdiction license with the same License Elements as this License.

d. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

e. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

f. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights

   Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

   Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

   b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

   c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

   d. to Distribute and Publicly Perform Adaptations.

   e. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   iii. **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

   The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions

   The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to

the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work

which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

   UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

   EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

## Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark “Creative Commons” or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons’ then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at http://creativecommons.org/.