The Components Book

Version: 3.0

generated on May 8, 2016

The Components Book (3.0)

This work is licensed under the "Attribution-Share Alike 3.0 Unported" license (*http://creativecommons.org/licenses/by-sa/3.0/*).

You are free **to share** (to copy, distribute and transmit the work), and **to remix** (to adapt the work) under the following conditions:

- **Attribution**: You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike**: If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor SensioLabs shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (*http://github.com/symfony/symfony-docs/issues*). Based on tickets and users feedback, this book is continuously updated.

Contents at a Glance

How to Install and Use the Symfony Components....6
The Asset Component....8
The BrowserKit Component....14
The ClassLoader Component....19
The PSR-0 Class Loader....20
The PSR-4 Class Loader....22
MapClassLoader....24
Cache a Class Loader....25
The Class Map Generator....27
Debugging a Class Loader....30
The Config Component....31
Loading Resources....32
Caching based on Resources....35
Defining and Processing Configuration Values....37
The Console Component....50
Using Console Commands, Shortcuts and Built-in Commands....60
Changing the Default Command....63
Building a single Command Application....65
Understanding how Console Arguments Are Handled....67
Using Events....69
Using the Logger....73
Dialog Helper....76
Formatter Helper....77
Process Helper....79
Progress Bar....82
Progress Helper....88
Question Helper....89
Table....95
Table Helper....100
Debug Formatter Helper....101
The CssSelector Component....104
The Debug Component....106
Debugging a Class Loader....108
The DependencyInjection Component....109
Types of Injection....114
Introduction to Parameters....117

Working with Container Service Definitions 121
Defining Services Dependencies Automatically (Autowiring) 125
How to Inject Instances into the Container 131
Compiling the Container 133
Working with Tagged Services 143
Using a Factory to Create Services 147
Configuring Services with a Service Configurator 149
Managing Common Dependencies with Parent Services 152
Advanced Container Configuration 156
Lazy Services 160
Container Building Workflow 162
The DomCrawler Component 164
The EventDispatcher Component 173
The Container Aware Event Dispatcher 181
The Generic Event Object 184
The Immutable Event Dispatcher 186
The Traceable Event Dispatcher 187
The ExpressionLanguage Component 189
The Expression Syntax 192
Extending the ExpressionLanguage 198
Caching Expressions Using Parser Caches 201
The Filesystem Component 203
LockHandler 209
The Finder Component 211
The Form Component 217
Creating a custom Type Guesser 228
Form Events 232
The HttpFoundation Component 241
Session Management 252
Configuring Sessions and Save Handlers 259
Testing with Sessions 265
Integrating with Legacy Sessions 267
Trusting Proxies 269
The HttpKernel Component 271
The Intl Component 288
The OptionsResolver Component 295
The PHPUnit Bridge 308
The Process Component 313
The PropertyAccess Component 319
The Routing Component 327
How to Match a Route Based on the Host 334
The Security Component 337
The Firewall and Authorization 339
Authentication 342
Authorization 348
Securely Generating Random Values 353
The Serializer Component 355

The Stopwatch Component....................367
The Templating Component....................370
Slots Helper....................374
Assets Helper....................376
The Translation Component....................379
Using the Translator....................383
Adding Custom Format Support....................389
The VarDumper Component....................392
Advanced Usage of the VarDumper Component....................398
The Yaml Component....................403
The YAML Format....................407

Chapter 1

How to Install and Use the Symfony Components

If you're starting a new project (or already have a project) that will use one or more components, the easiest way to integrate everything is with *Composer*[1]. Composer is smart enough to download the component(s) that you need and take care of autoloading so that you can begin using the libraries immediately.

This article will take you through using *The Finder Component*, though this applies to using any component.

Using the Finder Component

1. If you're creating a new project, create a new empty directory for it.

2. Open a terminal and use Composer to grab the library.

Listing 1-1

```
1  $ composer require symfony/finder
```

The name `symfony/finder` is written at the top of the documentation for whatever component you want.

Install composer[2] if you don't have it already present on your system. Depending on how you install, you may end up with a `composer.phar` file in your directory. In that case, no worries! Just run `php composer.phar require symfony/finder`.

3. Write your code!

Once Composer has downloaded the component(s), all you need to do is include the `vendor/autoload.php` file that was generated by Composer. This file takes care of autoloading all of the libraries so that you can use them immediately:

1. `https://getcomposer.org`
2. `https://getcomposer.org/download/`

Listing 1-2

```
// File example: src/script.php

// update this to the path to the "vendor/"
// directory, relative to this file
require_once __DIR__.'/../vendor/autoload.php';

use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->in('../data/');

// ...
```

Using all of the Components

If you want to use all of the Symfony Components, then instead of adding them one by one, you can include the `symfony/symfony` package:

Listing 1-3

```
$ composer require symfony/symfony
```

This will also include the Bundle and Bridge libraries, which you may or may not actually need.

Now what?

Now that the component is installed and autoloaded, read the specific component's documentation to find out more about how to use it.

And have fun!

Chapter 2
The Asset Component

> The Asset component manages URL generation and versioning of web assets such as CSS stylesheets, JavaScript files and image files.

In the past, it was common for web applications to hardcode URLs of web assets. For example:

Listing 2-1
```
<link rel="stylesheet" type="text/css" href="/css/main.css">

<!-- ... -->

<a href="/"><img src="/images/logo.png"></a>
```

This practice is no longer recommended unless the web application is extremely simple. Hardcoding URLs can be a disadvantage because:

- **Templates get verbose**: you have to write the full path for each asset. When using the Asset component, you can group assets in packages to avoid repeating the common part of their path;
- **Versioning is difficult**: it has to be custom managed for each application. Adding a version (e.g. `main.css?v=5`) to the asset URLs is essential for some applications because it allows you to control how the assets are cached. The Asset component allows you to define different versioning strategies for each package;
- **Moving assets location** is cumbersome and error-prone: it requires you to carefully update the URLs of all assets included in all templates. The Asset component allows to move assets effortlessly just by changing the base path value associated with the package of assets;
- **It's nearly impossible to use multiple CDNs**: this technique requires you to change the URL of the asset randomly for each request. The Asset component provides out-of-the-box support for any number of multiple CDNs, both regular (`http://`) and secure (`https://`).

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/asset` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/asset*).

Usage

Asset Packages

The Asset component manages assets through packages. A package groups all the assets which share the same properties: versioning strategy, base path, CDN hosts, etc. In the following basic example, a package is created to manage assets without any versioning:

Listing 2-2
```
1 use Symfony\Component\Asset\Package;
2 use Symfony\Component\Asset\VersionStrategy\EmptyVersionStrategy;
3
4 $package = new Package(new EmptyVersionStrategy());
5
6 echo $package->getUrl('/image.png');
7 // result: /image.png
```

Packages implement *`PackageInterface`*[2], which defines the following two methods:

`getVersion()`[3]
: Returns the asset version for an asset.

`getUrl()`[4]
: Returns an absolute or root-relative public path.

With a package, you can:

1. version the assets;
2. set a common base path (e.g. `/css`) for the assets;
3. configure a CDN for the assets

Versioned Assets

One of the main features of the Asset component is the ability to manage the versioning of the application's assets. Asset versions are commonly used to control how these assets are cached.

Instead of relying on a simple version mechanism, the Asset component allows you to define advanced versioning strategies via PHP classes. The two built-in strategies are the *`EmptyVersionStrategy`*[5], which doesn't add any version to the asset and *`StaticVersionStrategy`*[6], which allows you to set the version with a format string.

In this example, the `StaticVersionStrategy` is used to append the `v1` suffix to any asset path:

Listing 2-3
```
1 use Symfony\Component\Asset\Package;
2 use Symfony\Component\Asset\VersionStrategy\StaticVersionStrategy;
3
4 $package = new Package(new StaticVersionStrategy('v1'));
5
```

1. https://packagist.org/packages/symfony/asset
2. http://api.symfony.com/3.0/Symfony/Component/Asset/PackageInterface.html
3. http://api.symfony.com/3.0/Symfony/Component/Asset/PackageInterface.html#method_getVersion
4. http://api.symfony.com/3.0/Symfony/Component/Asset/PackageInterface.html#method_getUrl
5. http://api.symfony.com/3.0/Symfony/Component/Asset/VersionStrategy/EmptyVersionStrategy.html
6. http://api.symfony.com/3.0/Symfony/Component/Asset/VersionStrategy/StaticVersionStrategy.html

```
6 echo $package->getUrl('/image.png');
7 // result: /image.png?v1
```

In case you want to modify the version format, pass a sprintf-compatible format string as the second argument of the `StaticVersionStrategy` constructor:

Listing 2-4

```
1  // put the 'version' word before the version value
2  $package = new Package(new StaticVersionStrategy('v1', '%s?version=%s'));
3
4  echo $package->getUrl('/image.png');
5  // result: /image.png?version=v1
6
7  // put the asset version before its path
8  $package = new Package(new StaticVersionStrategy('v1', '%2$s/%1$s'));
9
10 echo $package->getUrl('/image.png');
11 // result: /v1/image.png
```

Custom Version Strategies

Use the *VersionStrategyInterface*[7] to define your own versioning strategy. For example, your application may need to append the current date to all its web assets in order to bust the cache every day:

Listing 2-5

```
1  use Symfony\Component\Asset\VersionStrategy\VersionStrategyInterface;
2
3  class DateVersionStrategy implements VersionStrategyInterface
4  {
5      private $version;
6
7      public function __construct()
8      {
9          $this->version = date('Ymd');
10     }
11
12     public function getVersion($path)
13     {
14         return $this->version;
15     }
16
17     public function applyVersion($path)
18     {
19         return sprintf('%s?v=%s', $path, $this->getVersion($path));
20     }
21 }
```

Grouped Assets

Often, many assets live under a common path (e.g. `/static/images`). If that's your case, replace the default *Package*[8] class with *PathPackage*[9] to avoid repeating that path over and over again:

Listing 2-6

7. http://api.symfony.com/3.0/Symfony/Component/Asset/VersionStrategy/VersionStrategyInterface.html
8. http://api.symfony.com/3.0/Symfony/Component/Asset/Package.html
9. http://api.symfony.com/3.0/Symfony/Component/Asset/PathPackage.html

```
1 use Symfony\Component\Asset\PathPackage;
2 // ...
3
4 $package = new PathPackage('/static/images', new StaticVersionStrategy('v1'));
5
6 echo $package->getUrl('/logo.png');
7 // result: /static/images/logo.png?v1
```

Request Context Aware Assets

If you are also using the *HttpFoundation* component in your project (for instance, in a Symfony application), the `PathPackage` class can take into account the context of the current request:

Listing 2-7

```
1 use Symfony\Component\Asset\PathPackage;
2 use Symfony\Component\Asset\Context\RequestStackContext;
3 // ...
4
5 $package = new PathPackage(
6     '/static/images',
7     new StaticVersionStrategy('v1'),
8     new RequestStackContext($requestStack)
9 );
10
11 echo $package->getUrl('/logo.png');
12 // result: /somewhere/static/images/logo.png?v1
```

Now that the request context is set, the `PathPackage` will prepend the current request base URL. So, for example, if your entire site is hosted under the `/somewhere` directory of your web server root directory and the configured base path is `/static/images`, all paths will be prefixed with `/somewhere/static/images`.

Absolute Assets and CDNs

Applications that host their assets on different domains and CDNs (*Content Delivery Networks*) should use the *`UrlPackage`*[10] class to generate absolute URLs for their assets:

Listing 2-8

```
1 use Symfony\Component\Asset\UrlPackage;
2 // ...
3
4 $package = new UrlPackage(
5     'http://static.example.com/images/',
6     new StaticVersionStrategy('v1')
7 );
8
9 echo $package->getUrl('/logo.png');
10 // result: http://static.example.com/images/logo.png?v1
```

You can also pass a schema-agnostic URL:

Listing 2-9

```
1 use Symfony\Component\Asset\UrlPackage;
2 // ...
3
4 $package = new UrlPackage(
```

10. http://api.symfony.com/3.0/Symfony/Component/Asset/UrlPackage.html

```
5     '//static.example.com/images/',
6     new StaticVersionStrategy('v1')
7 );
8
9 echo $package->getUrl('/logo.png');
10 // result: //static.example.com/images/logo.png?v1
```

This is useful because assets will automatically be requested via HTTPS if a visitor is viewing your site in https. Just make sure that your CDN host supports https.

In case you serve assets from more than one domain to improve application performance, pass an array of URLs as the first argument to the `UrlPackage` constructor:

Listing 2-10

```
1 use Symfony\Component\Asset\UrlPackage;
2 // ...
3
4 $urls = array(
5     '//static1.example.com/images/',
6     '//static2.example.com/images/',
7 );
8 $package = new UrlPackage($urls, new StaticVersionStrategy('v1'));
9
10 echo $package->getUrl('/logo.png');
11 // result: http://static1.example.com/images/logo.png?v1
12 echo $package->getUrl('/icon.png');
13 // result: http://static2.example.com/images/icon.png?v1
```

For each asset, one of the URLs will be randomly used. But, the selection is deterministic, meaning that each asset will be always served by the same domain. This behavior simplifies the management of HTTP cache.

Request Context Aware Assets

Similarly to application-relative assets, absolute assets can also take into account the context of the current request. In this case, only the request scheme is considered, in order to select the appropriate base URL (HTTPs or protocol-relative URLs for HTTPs requests, any base URL for HTTP requests):

Listing 2-11

```
1 use Symfony\Component\Asset\UrlPackage;
2 use Symfony\Component\Asset\Context\RequestStackContext;
3 // ...
4
5 $package = new UrlPackage(
6     array('http://example.com/', 'https://example.com/'),
7     new StaticVersionStrategy('v1'),
8     new RequestStackContext($requestStack)
9 );
10
11 echo $package->getUrl('/logo.png');
12 // assuming the RequestStackContext says that we are on a secure host
13 // result: https://example.com/logo.png?v1
```

Named Packages

Applications that manage lots of different assets may need to group them in packages with the same versioning strategy and base path. The Asset component includes a *`Packages`*[11] class to simplify management of several packages.

In the following example, all packages use the same versioning strategy, but they all have different base paths:

Listing 2-12
```
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\PathPackage;
use Symfony\Component\Asset\UrlPackage;
use Symfony\Component\Asset\Packages;
// ...

$versionStrategy = new StaticVersionStrategy('v1');

$defaultPackage = new Package($versionStrategy);

$namedPackages = array(
    'img' => new UrlPackage('http://img.example.com/', $versionStrategy),
    'doc' => new PathPackage('/somewhere/deep/for/documents', $versionStrategy),
);

$packages = new Packages($defaultPackage, $namedPackages)
```

The `Packages` class allows to define a default package, which will be applied to assets that don't define the name of package to use. In addition, this application defines a package named `img` to serve images from an external domain and a `doc` package to avoid repeating long paths when linking to a document inside a template:

Listing 2-13
```
echo $packages->getUrl('/main.css');
// result: /main.css?v1

echo $packages->getUrl('/logo.png', 'img');
// result: http://img.example.com/logo.png?v1

echo $packages->getUrl('/resume.pdf', 'doc');
// result: /somewhere/deep/for/documents/resume.pdf?v1
```

11. http://api.symfony.com/3.0/Symfony/Component/Asset/Packages.html

Chapter 3
The BrowserKit Component

The BrowserKit component simulates the behavior of a web browser, allowing you to make requests, click on links and submit forms programmatically.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/browser-kit` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/browser-kit*).

Basic Usage

Creating a Client

The component only provides an abstract client and does not provide any backend ready to use for the HTTP layer.

To create your own client, you must extend the abstract `Client` class and implement the *`doRequest()`*[2] method. This method accepts a request and should return a response:

Listing 3-1

```
namespace Acme;

use Symfony\Component\BrowserKit\Client as BaseClient;
use Symfony\Component\BrowserKit\Response;

class Client extends BaseClient
{
```

1. https://packagist.org/packages/symfony/browser-kit
2. http://api.symfony.com/3.0/Symfony/Component/BrowserKit/Client.html#method_doRequest

```
 8        protected function doRequest($request)
 9        {
10            // ... convert request into a response
11
12            return new Response($content, $status, $headers);
13        }
14    }
```

For a simple implementation of a browser based on the HTTP layer, have a look at *Goutte*[3]. For an implementation based on `HttpKernelInterface`, have a look at the *`Client`*[4] provided by the *HttpKernel component*.

Making Requests

Use the *`request()`*[5] method to make HTTP requests. The first two arguments are the HTTP method and the requested URL:

Listing 3-2

```
use Acme\Client;

$client = new Client();
$crawler = $client->request('GET', 'http://symfony.com');
```

The value returned by the `request()` method is an instance of the *`Crawler`*[6] class, provided by the *DomCrawler component*, which allows accessing and traversing HTML elements programmatically.

Clicking Links

The `Crawler` object is capable of simulating link clicks. First, pass the text content of the link to the `selectLink()` method, which returns a `Link` object. Then, pass this object to the `click()` method, which performs the needed HTTP GET request to simulate the link click:

Listing 3-3

```
1  use Acme\Client;
2
3  $client = new Client();
4  $crawler = $client->request('GET', 'http://symfony.com');
5  $link = $crawler->selectLink('Go elsewhere...')->link();
6  $client->click($link);
```

Submitting Forms

The `Crawler` object is also capable of selecting forms. First, select any of the form's buttons with the `selectButton()` method. Then, use the `form()` method to select the form which the button belongs to.

After selecting the form, fill in its data and send it using the `submit()` method (which makes the needed HTTP POST request to submit the form contents):

Listing 3-4

```
1  use Acme\Client;
2
3  // make a real request to an external site
4  $client = new Client();
```

3. https://github.com/fabpot/Goutte
4. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Client.html
5. http://api.symfony.com/3.0/Symfony/Component/BrowserKit/Client.html#method_request
6. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html

```
5   $crawler = $client->request('GET', 'https://github.com/login');
6
7   // select the form and fill in some values
8   $form = $crawler->selectButton('Log in')->form();
9   $form['login'] = 'symfonyfan';
10  $form['password'] = 'anypass';
11
12  // submit that form
13  $crawler = $client->submit($form);
```

Cookies

Retrieving Cookies

The `Crawler` object exposes cookies (if any) through a *`CookieJar`*[7], which allows you to store and retrieve any cookie while making requests with the client:

Listing 3-5

```
1   use Acme\Client;
2
3   // Make a request
4   $client = new Client();
5   $crawler = $client->request('GET', 'http://symfony.com');
6
7   // Get the cookie Jar
8   $cookieJar = $crawler->getCookieJar();
9
10  // Get a cookie by name
11  $cookie = $cookieJar->get('name_of_the_cookie');
12
13  // Get cookie data
14  $name       = $cookie->getName();
15  $value      = $cookie->getValue();
16  $raw        = $cookie->getRawValue();
17  $secure     = $cookie->isSecure();
18  $isHttpOnly = $cookie->isHttpOnly();
19  $isExpired  = $cookie->isExpired();
20  $expires    = $cookie->getExpiresTime();
21  $path       = $cookie->getPath();
22  $domain     = $cookie->getDomain();
```

These methods only return cookies that have not expired.

Looping Through Cookies

Listing 3-6

```
1   use Acme\Client;
2
3   // Make a request
```

7. http://api.symfony.com/3.0/Symfony/Component/BrowserKit/CookieJar.html

```
4 $client = new Client();
5 $crawler = $client->request('GET', 'http://symfony.com');
6
7 // Get the cookie Jar
8 $cookieJar = $crawler->getCookieJar();
9
10 // Get array with all cookies
11 $cookies = $cookieJar->all();
12 foreach ($cookies as $cookie) {
13     // ...
14 }
15
16 // Get all values
17 $values = $cookieJar->allValues('http://symfony.com');
18 foreach ($values as $value) {
19     // ...
20 }
21
22 // Get all raw values
23 $rawValues = $cookieJar->allRawValues('http://symfony.com');
24 foreach ($rawValues as $rawValue) {
25     // ...
26 }
```

Setting Cookies

You can also create cookies and add them to a cookie jar that can be injected into the client constructor:

Listing 3-7

```
1 use Acme\Client;
2
3 // create cookies and add to cookie jar
4 $cookieJar = new Cookie('flavor', 'chocolate', strtotime('+1 day'));
5
6 // create a client and set the cookies
7 $client = new Client(array(), array(), $cookieJar);
8 // ...
```

History

The client stores all your requests allowing you to go back and forward in your history:

Listing 3-8

```
1 use Acme\Client;
2
3 // make a real request to an external site
4 $client = new Client();
5 $client->request('GET', 'http://symfony.com');
6
7 // select and click on a link
8 $link = $crawler->selectLink('Documentation')->link();
9 $client->click($link);
10
11 // go back to home page
12 $crawler = $client->back();
13
```

```
14 // go forward to documentation page
15 $crawler = $client->forward();
```

You can delete the client's history with the `restart()` method. This will also delete all the cookies:

Listing 3-9

```
1 use Acme\Client;
2
3 // make a real request to an external site
4 $client = new Client();
5 $client->request('GET', 'http://symfony.com');
6
7 // delete history
8 $client->restart();
```

Chapter 4

The ClassLoader Component

> The ClassLoader component provides tools to autoload your classes and cache their locations for performance.

Usage

Whenever you reference a class that has not been required or included yet, PHP uses the *autoloading mechanism*[1] to delegate the loading of a file defining the class. Symfony provides three autoloaders, which are able to load your classes:

- *The PSR-0 Class Loader*: loads classes that follow the *PSR-0*[2] class naming standard;
- *The PSR-4 Class Loader*: loads classes that follow the *PSR-4* class naming standard;
- *MapClassLoader*: loads classes using a static map from class name to file path.

Additionally, the Symfony ClassLoader component ships with a wrapper class which makes it possible *to cache the results of a class loader*.

When using the *Debug component*, you can also use a special *DebugClassLoader* that eases debugging by throwing more helpful exceptions when a class could not be found by a class loader.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/class-loader` on *Packagist*[3]);
- Use the official Git repository (*https://github.com/symfony/class-loader*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

1. `http://php.net/manual/en/language.oop5.autoload.php`
2. `http://www.php-fig.org/psr/psr-0/`
3. `https://packagist.org/packages/symfony/class-loader`

Chapter 5

The PSR-0 Class Loader

If your classes and third-party libraries follow the *PSR-0*[1] standard, you can use the *ClassLoader*[2] class to load all of your project's classes.

You can use both the ApcClassLoader and the XcacheClassLoader to *cache* a ClassLoader instance.

Usage

Registering the *ClassLoader*[3] autoloader is straightforward:

Listing 5-1

```
require_once '/path/to/src/Symfony/Component/ClassLoader/ClassLoader.php';

use Symfony\Component\ClassLoader\ClassLoader;

$loader = new ClassLoader();

// to enable searching the include path (eg. for PEAR packages)
$loader->setUseIncludePath(true);

// ... register namespaces and prefixes here - see below

$loader->register();
```

The autoloader is automatically registered in a Symfony application (see app/autoload.php).

1. http://www.php-fig.org/psr/psr-0/
2. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassLoader.html
3. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassLoader.html

Use *addPrefix()*[4] or *addPrefixes()*[5] to register your classes:

Listing 5-2

```
// register a single namespaces
$loader->addPrefix('Symfony', __DIR__.'/vendor/symfony/symfony/src');

// register several namespaces at once
$loader->addPrefixes(array(
    'Symfony' => __DIR__.'/../vendor/symfony/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/monolog/src',
));

// register a prefix for a class following the PEAR naming conventions
$loader->addPrefix('Twig_', __DIR__.'/vendor/twig/twig/lib');

$loader->addPrefixes(array(
    'Swift_' => __DIR__.'/vendor/swiftmailer/swiftmailer/lib/classes',
    'Twig_'  => __DIR__.'/vendor/twig/twig/lib',
));
```

Classes from a sub-namespace or a sub-hierarchy of *PEAR*[6] classes can be looked for in a location list to ease the vendoring of a sub-set of classes for large projects:

Listing 5-3

```
$loader->addPrefixes(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine/common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine/migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine/dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/orm/lib',
));
```

In this example, if you try to use a class in the `Doctrine\Common` namespace or one of its children, the autoloader will first look for the class under the `doctrine-common` directory. If not found, it will then fallback to the default `Doctrine` directory (the last one configured) before giving up. The order of the prefix registrations is significant in this case.

4. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassLoader.html#method_addPrefix
5. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassLoader.html#method_addPrefixes
6. http://pear.php.net/manual/en/standards.naming.php

Chapter 6

The PSR-4 Class Loader

Libraries that follow the *PSR-4*[1] standard can be loaded with the `Psr4ClassLoader`.

If you manage your dependencies via Composer, you get a PSR-4 compatible autoloader out of the box. Use this loader in environments where Composer is not available.

All Symfony components follow PSR-4.

Usage

The following example demonstrates how you can use the *`Psr4ClassLoader`*[2] autoloader to use Symfony's Yaml component. Imagine, you downloaded both the ClassLoader and Yaml component as ZIP packages and unpacked them to a `libs` directory. The directory structure will look like this:

Listing 6-1

```
libs/
    ClassLoader/
        Psr4ClassLoader.php
        ...
    Yaml/
        Yaml.php
        ...
config.yml
demo.php
```

In `demo.php` you are going to parse the `config.yml` file. To do that, you first need to configure the `Psr4ClassLoader`:

1. http://www.php-fig.org/psr/psr-4/
2. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/Psr4ClassLoader.html

Listing 6-2

```
use Symfony\Component\ClassLoader\Psr4ClassLoader;
use Symfony\Component\Yaml\Yaml;

require __DIR__.'/lib/ClassLoader/Psr4ClassLoader.php';

$loader = new Psr4ClassLoader();
$loader->addPrefix('Symfony\\Component\\Yaml\\', __DIR__.'/lib/Yaml');
$loader->register();

$data = Yaml::parse(file_get_contents(__DIR__.'/config.yml'));
```

First of all, the class loader is loaded manually using a `require` statement, since there is no autoload mechanism yet. With the *`addPrefix()`*[3] call, you tell the class loader where to look for classes with the `Symfony\Component\Yaml\` namespace prefix. After registering the autoloader, the Yaml component is ready to be used.

3. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/Psr4ClassLoader.html#method_addPrefix

Chapter 7
MapClassLoader

The *MapClassLoader*[1] allows you to autoload files via a static map from classes to files. This is useful if you use third-party libraries which don't follow the *PSR-0*[2] standards and so can't use the *PSR-0 class loader*.

The `MapClassLoader` can be used along with the *PSR-0 class loader* by configuring and calling the `register()` method on both.

The default behavior is to append the `MapClassLoader` on the autoload stack. If you want to use it as the first autoloader, pass `true` when calling the `register()` method. Your class loader will then be prepended on the autoload stack.

Usage

Using it is as easy as passing your mapping to its constructor when creating an instance of the `MapClassLoader` class:

Listing 7-1

```
require_once '/path/to/src/Symfony/Component/ClassLoader/MapClassLoader.php';

$mapping = array(
    'Foo' => '/path/to/Foo',
    'Bar' => '/path/to/Bar',
);

$loader = new MapClassLoader($mapping);

$loader->register();
```

1. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/MapClassLoader.html
2. http://www.php-fig.org/psr/psr-0/

Chapter 8

Cache a Class Loader

Introduction

Finding the file for a particular class can be an expensive task. Luckily, the ClassLoader component comes with two classes to cache the mapping from a class to its containing file. Both the *ApcClassLoader*[1] and the *XcacheClassLoader*[2] wrap around an object which implements a `findFile()` method to find the file for a class.

Both the `ApcClassLoader` and the `XcacheClassLoader` can be used to cache Composer's *autoloader*[3].

ApcClassLoader

`ApcClassLoader` wraps an existing class loader and caches calls to its `findFile()` method using *APC*[4]:

Listing 8-1

```
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcClassLoader.php';

// instance of a class that implements a findFile() method, like the ClassLoader
$loader = ...;

// sha1(__FILE__) generates an APC namespace prefix
$cachedLoader = new ApcClassLoader(sha1(__FILE__), $loader);

// register the cached class loader
$cachedLoader->register();
```

1. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ApcClassLoader.html
2. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/XcacheClassLoader.html
3. https://getcomposer.org/doc/01-basic-usage.md#autoloading
4. http://php.net/manual/en/book.apc.php

```
11
12 // deactivate the original, non-cached loader if it was registered previously
13 $loader->unregister();
```

XcacheClassLoader

`XcacheClassLoader` uses *XCache*[5] to cache a class loader. Registering it is straightforward:

Listing 8-2

```
1  require_once '/path/to/src/Symfony/Component/ClassLoader/XcacheClassLoader.php';
2
3  // instance of a class that implements a findFile() method, like the ClassLoader
4  $loader = ...;
5
6  // sha1(__FILE__) generates an XCache namespace prefix
7  $cachedLoader = new XcacheClassLoader(sha1(__FILE__), $loader);
8
9  // register the cached class loader
10 $cachedLoader->register();
11
12 // deactivate the original, non-cached loader if it was registered previously
13 $loader->unregister();
```

5. `http://xcache.lighttpd.net`

Chapter 9
The Class Map Generator

Loading a class usually is an easy task given the *PSR-0*[1] and *PSR-4*[2] standards. Thanks to the Symfony ClassLoader component or the autoloading mechanism provided by Composer, you don't have to map your class names to actual PHP files manually. Nowadays, PHP libraries usually come with autoloading support through Composer.

But from time to time you may have to use a third-party library that comes without any autoloading support and therefore forces you to load each class manually. For example, imagine a library with the following directory structure:

Listing 9-1
```
library/
├── bar/
│   ├── baz/
│   │   └── Boo.php
│   └── Foo.php
└── foo/
    ├── bar/
    │   └── Foo.php
    └── Bar.php
```

These files contain the following classes:

File	Class Name
`library/bar/baz/Boo.php`	`Acme\Bar\Baz`
`library/bar/Foo.php`	`Acme\Bar`
`library/foo/bar/Foo.php`	`Acme\Foo\Bar`
`library/foo/Bar.php`	`Acme\Foo`

To make your life easier, the ClassLoader component comes with a *`ClassMapGenerator`*[3] class that makes it possible to create a map of class names to files.

1. http://www.php-fig.org/psr/psr-0
2. http://www.php-fig.org/psr/psr-4
3. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassMapGenerator.html

Generating a Class Map

To generate the class map, simply pass the root directory of your class files to the *createMap()*[4] method:

Listing 9-2
```
use Symfony\Component\ClassLoader\ClassMapGenerator;

var_dump(ClassMapGenerator::createMap(__DIR__.'/library'));
```

Given the files and class from the table above, you should see an output like this:

Listing 9-3
```
1  Array
2  (
3      [Acme\Foo] => /var/www/library/foo/Bar.php
4      [Acme\Foo\Bar] => /var/www/library/foo/bar/Foo.php
5      [Acme\Bar\Baz] => /var/www/library/bar/baz/Boo.php
6      [Acme\Bar] => /var/www/library/bar/Foo.php
7  )
```

Dumping the Class Map

Writing the class map to the console output is not really sufficient when it comes to autoloading. Luckily, the `ClassMapGenerator` provides the *dump()*[5] method to save the generated class map to the filesystem:

Listing 9-4
```
use Symfony\Component\ClassLoader\ClassMapGenerator;

ClassMapGenerator::dump(__DIR__.'/library', __DIR__.'/class_map.php');
```

This call to `dump()` generates the class map and writes it to the `class_map.php` file in the same directory with the following contents:

Listing 9-5
```
1  <?php return array (
2  'Acme\\Foo' => '/var/www/library/foo/Bar.php',
3  'Acme\\Foo\\Bar' => '/var/www/library/foo/bar/Foo.php',
4  'Acme\\Bar\\Baz' => '/var/www/library/bar/baz/Boo.php',
5  'Acme\\Bar' => '/var/www/library/bar/Foo.php',
6  );
```

Instead of loading each file manually, you'll only have to register the generated class map with, for example, the *MapClassLoader*[6]:

Listing 9-6
```
1   use Symfony\Component\ClassLoader\MapClassLoader;
2
3   $mapping = include __DIR__.'/class_map.php';
4   $loader = new MapClassLoader($mapping);
5   $loader->register();
6
7   // you can now use the classes:
8   use Acme\Foo;
9
10  $foo = new Foo();
```

4. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassMapGenerator.html#method_createMap
5. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ClassMapGenerator.html#method_dump
6. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/MapClassLoader.html

```
11
12 // ...
```

The example assumes that you already have autoloading working (e.g. through *Composer*[7] or one of the other class loaders from the ClassLoader component.

Besides dumping the class map for one directory, you can also pass an array of directories for which to generate the class map (the result actually is the same as in the example above):

Listing 9-7

```
1 use Symfony\Component\ClassLoader\ClassMapGenerator;
2
3 ClassMapGenerator::dump(
4     array(__DIR__.'/library/bar', __DIR__.'/library/foo'),
5     __DIR__.'/class_map.php'
6 );
```

7. https://getcomposer.org

Chapter 10

Debugging a Class Loader

The `DebugClassLoader` from the ClassLoader component was deprecated in Symfony 2.5 and removed in Symfony 3.0. Use the *DebugClassLoader provided by the Debug component*.

Chapter 11
The Config Component

The Config component provides several classes to help you find, load, combine, autofill and validate configuration values of any kind, whatever their source may be (YAML, XML, INI files, or for instance a database).

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/config` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/config*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Sections

- *Loading Resources*
- *Caching based on Resources*
- *Defining and Processing Configuration Values*

1. `https://packagist.org/packages/symfony/config`

Chapter 12

Loading Resources

The `IniFileLoader` parses the file contents using the *parse_ini_file*[1] function. Therefore, you can only set parameters to string values. To set parameters to other data types (e.g. boolean, integer, etc), the other loaders are recommended.

Locating Resources

Loading the configuration normally starts with a search for resources, mostly files. This can be done with the *FileLocator*[2]:

Listing 12-1

```
use Symfony\Component\Config\FileLocator;

$configDirectories = array(__DIR__.'/app/config');

$locator = new FileLocator($configDirectories);
$yamlUserFiles = $locator->locate('users.yml', null, false);
```

The locator receives a collection of locations where it should look for files. The first argument of `locate()` is the name of the file to look for. The second argument may be the current path and when supplied, the locator will look in this directory first. The third argument indicates whether or not the locator should return the first file it has found or an array containing all matches.

1. http://php.net/manual/en/function.parse-ini-file.php
2. http://api.symfony.com/3.0/Symfony/Component/Config/FileLocator.html

Resource Loaders

For each type of resource (YAML, XML, annotation, etc.) a loader must be defined. Each loader should implement *LoaderInterface*[3] or extend the abstract *FileLoader*[4] class, which allows for recursively importing other resources:

Listing 12-2

```
use Symfony\Component\Config\Loader\FileLoader;
use Symfony\Component\Yaml\Yaml;

class YamlUserLoader extends FileLoader
{
    public function load($resource, $type = null)
    {
        $configValues = Yaml::parse(file_get_contents($resource));

        // ... handle the config values

        // maybe import some other resource:

        // $this->import('extra_users.yml');
    }

    public function supports($resource, $type = null)
    {
        return is_string($resource) && 'yml' === pathinfo(
            $resource,
            PATHINFO_EXTENSION
        );
    }
}
```

Finding the Right Loader

The *LoaderResolver*[5] receives as its first constructor argument a collection of loaders. When a resource (for instance an XML file) should be loaded, it loops through this collection of loaders and returns the loader which supports this particular resource type.

The *DelegatingLoader*[6] makes use of the *LoaderResolver*[7]. When it is asked to load a resource, it delegates this question to the *LoaderResolver*[8]. In case the resolver has found a suitable loader, this loader will be asked to load the resource:

Listing 12-3

```
use Symfony\Component\Config\Loader\LoaderResolver;
use Symfony\Component\Config\Loader\DelegatingLoader;

$loaderResolver = new LoaderResolver(array(new YamlUserLoader($locator)));
$delegatingLoader = new DelegatingLoader($loaderResolver);

$delegatingLoader->load(__DIR__.'/users.yml');
/*
```

3. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/FileLoader.html
5. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderResolver.html
6. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/DelegatingLoader.html
7. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderResolver.html
8. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderResolver.html

```
 9 The YamlUserLoader will be used to load this resource,
10 since it supports files with a "yml" extension
11 */
```

Chapter 13

Caching based on Resources

When all configuration resources are loaded, you may want to process the configuration values and combine them all in one file. This file acts like a cache. Its contents don't have to be regenerated every time the application runs – only when the configuration resources are modified.

For example, the Symfony Routing component allows you to load all routes, and then dump a URL matcher or a URL generator based on these routes. In this case, when one of the resources is modified (and you are working in a development environment), the generated file should be invalidated and regenerated. This can be accomplished by making use of the `ConfigCache`[1] class.

The example below shows you how to collect resources, then generate some code based on the resources that were loaded and write this code to the cache. The cache also receives the collection of resources that were used for generating the code. By looking at the "last modified" timestamp of these resources, the cache can tell if it is still fresh or that its contents should be regenerated:

Listing 13-1

```
use Symfony\Component\Config\ConfigCache;
use Symfony\Component\Config\Resource\FileResource;

$cachePath = __DIR__.'/cache/appUserMatcher.php';

// the second argument indicates whether or not you want to use debug mode
$userMatcherCache = new ConfigCache($cachePath, true);

if (!$userMatcherCache->isFresh()) {
    // fill this with an array of 'users.yml' file paths
    $yamlUserFiles = ...;

    $resources = array();

    foreach ($yamlUserFiles as $yamlUserFile) {
        // see the previous article "Loading resources" to
        // see where $delegatingLoader comes from
        $delegatingLoader->load($yamlUserFile);
        $resources[] = new FileResource($yamlUserFile);
    }
```

1. http://api.symfony.com/3.0/Symfony/Component/Config/ConfigCache.html

```
21
22     // the code for the UserMatcher is generated elsewhere
23     $code = ...;
24
25     $userMatcherCache->write($code, $resources);
26 }
27
28 // you may want to require the cached code:
29 require $cachePath;
```

In debug mode, a `.meta` file will be created in the same directory as the cache file itself. This `.meta` file contains the serialized resources, whose timestamps are used to determine if the cache is still fresh. When not in debug mode, the cache is considered to be "fresh" as soon as it exists, and therefore no `.meta` file will be generated.

Chapter 14

Defining and Processing Configuration Values

Validating Configuration Values

After loading configuration values from all kinds of resources, the values and their structure can be validated using the "Definition" part of the Config Component. Configuration values are usually expected to show some kind of hierarchy. Also, values should be of a certain type, be restricted in number or be one of a given set of values. For example, the following configuration (in YAML) shows a clear hierarchy and some validation rules that should be applied to it (like: "the value for `auto_connect` must be a boolean value"):

Listing 14-1

```
auto_connect: true
default_connection: mysql
connections:
    mysql:
        host:     localhost
        driver:   mysql
        username: user
        password: pass
    sqlite:
        host:     localhost
        driver:   sqlite
        memory:   true
        username: user
        password: pass
```

When loading multiple configuration files, it should be possible to merge and overwrite some values. Other values should not be merged and stay as they are when first encountered. Also, some keys are only available when another key has a specific value (in the sample configuration above: the `memory` key only makes sense when the `driver` is `sqlite`).

Defining a Hierarchy of Configuration Values Using the TreeBuilder

All the rules concerning configuration values can be defined using the *TreeBuilder*[1].

A *TreeBuilder*[2] instance should be returned from a custom Configuration class which implements the *ConfigurationInterface*[3]:

Listing 14-2

```
namespace Acme\DatabaseConfiguration;

use Symfony\Component\Config\Definition\ConfigurationInterface;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;

class DatabaseConfiguration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('database');

        // ... add node definitions to the root of the tree

        return $treeBuilder;
    }
}
```

Adding Node Definitions to the Tree

Variable Nodes

A tree contains node definitions which can be laid out in a semantic way. This means, using indentation and the fluent notation, it is possible to reflect the real structure of the configuration values:

Listing 14-3

```
$rootNode
    ->children()
        ->booleanNode('auto_connect')
            ->defaultTrue()
        ->end()
        ->scalarNode('default_connection')
            ->defaultValue('default')
        ->end()
    ->end()
;
```

The root node itself is an array node, and has children, like the boolean node auto_connect and the scalar node default_connection. In general: after defining a node, a call to end() takes you one step up in the hierarchy.

Node Type

It is possible to validate the type of a provided value by using the appropriate node definition. Node types are available for:

1. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
2. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
3. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/ConfigurationInterface.html

- scalar (generic type that includes booleans, strings, integers, floats and `null`)
- boolean
- integer
- float
- enum (similar to scalar, but it only allows a finite set of values)
- array
- variable (no validation)

and are created with `node($name, $type)` or their associated shortcut `xxxxNode($name)` method.

Numeric Node Constraints

Numeric nodes (float and integer) provide two extra constraints - *`min()`*[4] and *`max()`*[5] - allowing to validate the value:

Listing 14-4

```
$rootNode
    ->children()
        ->integerNode('positive_value')
            ->min(0)
        ->end()
        ->floatNode('big_value')
            ->max(5E45)
        ->end()
        ->integerNode('value_inside_a_range')
            ->min(-50)->max(50)
        ->end()
    ->end()
;
```

Enum Nodes

Enum nodes provide a constraint to match the given input against a set of values:

Listing 14-5

```
$rootNode
    ->children()
        ->enumNode('gender')
            ->values(array('male', 'female'))
        ->end()
    ->end()
;
```

This will restrict the `gender` option to be either `male` or `female`.

Array Nodes

It is possible to add a deeper level to the hierarchy, by adding an array node. The array node itself, may have a pre-defined set of variable nodes:

Listing 14-6

```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
```

4. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/IntegerNodeDefinition.html#method_min
5. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/IntegerNodeDefinition.html#method_max

```
5                   ->scalarNode('driver')->end()
6                   ->scalarNode('host')->end()
7                   ->scalarNode('username')->end()
8                   ->scalarNode('password')->end()
9               ->end()
10          ->end()
11      ->end()
12  ;
```

Or you may define a prototype for each node inside an array node:

Listing 14-7

```
1  $rootNode
2      ->children()
3          ->arrayNode('connections')
4              ->prototype('array')
5                  ->children()
6                      ->scalarNode('driver')->end()
7                      ->scalarNode('host')->end()
8                      ->scalarNode('username')->end()
9                      ->scalarNode('password')->end()
10                 ->end()
11             ->end()
12         ->end()
13     ->end()
14 ;
```

A prototype can be used to add a definition which may be repeated many times inside the current node. According to the prototype definition in the example above, it is possible to have multiple connection arrays (containing a `driver`, `host`, etc.).

Array Node Options

Before defining the children of an array node, you can provide options like:

`useAttributeAsKey()`
: Provide the name of a child node, whose value should be used as the key in the resulting array. This method also defines the way config array keys are treated, as explained in the following example.

`requiresAtLeastOneElement()`
: There should be at least one element in the array (works only when `isRequired()` is also called).

`addDefaultsIfNotSet()`
: If any child nodes have default values, use them if explicit values haven't been provided.

`normalizeKeys(false)`
: If called (with `false`), keys with dashes are *not* normalized to underscores. It is recommended to use this with prototype nodes where the user will define a key-value map, to avoid an unnecessary transformation.

A basic prototyped array configuration can be defined as follows:

Listing 14-8

```
1  $node
2      ->fixXmlConfig('driver')
3      ->children()
4          ->arrayNode('drivers')
5              ->prototype('scalar')->end()
```

```
6             ->end()
7         ->end()
8 ;
```

When using the following YAML configuration:

Listing 14-9
```
1 drivers: ['mysql', 'sqlite']
```

Or the following XML configuration:

Listing 14-10
```
1 <driver>mysql</driver>
2 <driver>sqlite</driver>
```

The processed configuration is:

Listing 14-11
```
Array(
    [0] => 'mysql'
    [1] => 'sqlite'
)
```

A more complex example would be to define a prototyped array with children:

Listing 14-12
```
 1 $node
 2     ->fixXmlConfig('connection')
 3     ->children()
 4         ->arrayNode('connections')
 5             ->prototype('array')
 6                 ->children()
 7                     ->scalarNode('table')->end()
 8                     ->scalarNode('user')->end()
 9                     ->scalarNode('password')->end()
10                 ->end()
11             ->end()
12         ->end()
13     ->end()
14 ;
```

When using the following YAML configuration:

Listing 14-13
```
1 connections:
2     - { table: symfony, user: root, password: ~ }
3     - { table: foo, user: root, password: pa$$ }
```

Or the following XML configuration:

Listing 14-14
```
1 <connection table="symfony" user="root" password="null" />
2 <connection table="foo" user="root" password="pa$$" />
```

The processed configuration is:

Listing 14-15
```
1 Array(
2     [0] => Array(
3         [table] => 'symfony'
4         [user] => 'root'
```

```
5             [password] => null
6         )
7         [1] => Array(
8             [table] => 'foo'
9             [user] => 'root'
10            [password] => 'pa$$'
11        )
12    )
```

The previous output matches the expected result. However, given the configuration tree, when using the following YAML configuration:

Listing 14-16
```
1  connections:
2      sf_connection:
3          table: symfony
4          user: root
5          password: ~
6      default:
7          table: foo
8          user: root
9          password: pa$$
```

The output configuration will be exactly the same as before. In other words, the `sf_connection` and `default` configuration keys are lost. The reason is that the Symfony Config component treats arrays as lists by default.

As of writing this, there is an inconsistency: if only one file provides the configuration in question, the keys (i.e. `sf_connection` and `default`) are *not* lost. But if more than one file provides the configuration, the keys are lost as described above.

In order to maintain the array keys use the `useAttributeAsKey()` method:

Listing 14-17
```
1  $node
2      ->fixXmlConfig('connection')
3      ->children()
4          ->arrayNode('connections')
5              ->useAttributeAsKey('name')
6              ->prototype('array')
7                  ->children()
8                      ->scalarNode('table')->end()
9                      ->scalarNode('user')->end()
10                     ->scalarNode('password')->end()
11                 ->end()
12             ->end()
13         ->end()
14     ->end()
15 ;
```

The argument of this method (`name` in the example above) defines the name of the attribute added to each XML node to differentiate them. Now you can use the same YAML configuration shown before or the following XML configuration:

Listing 14-18
```
1  <connection name="sf_connection"
2      table="symfony" user="root" password="null" />
```

```
3 <connection name="default"
4     table="foo" user="root" password="pa$$" />
```

In both cases, the processed configuration maintains the `sf_connection` and `default` keys:

Listing 14-19

```
Array(
    [sf_connection] => Array(
        [table] => 'symfony'
        [user] => 'root'
        [password] => null
    )
    [default] => Array(
        [table] => 'foo'
        [user] => 'root'
        [password] => 'pa$$'
    )
)
```

Default and Required Values

For all node types, it is possible to define default values and replacement values in case a node has a certain value:

`defaultValue()`
: Set a default value

`isRequired()`
: Must be defined (but may be empty)

`cannotBeEmpty()`
: May not contain an empty value

`default*()`
: (`null`, `true`, `false`), shortcut for `defaultValue()`

`treat*Like()`
: (`null`, `true`, `false`), provide a replacement value in case the value is `*.`

Listing 14-20

```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->cannotBeEmpty()
                ->end()
                ->scalarNode('host')
                    ->defaultValue('localhost')
                ->end()
                ->scalarNode('username')->end()
                ->scalarNode('password')->end()
                ->booleanNode('memory')
                    ->defaultFalse()
                ->end()
```

```
17                ->end()
18            ->end()
19            ->arrayNode('settings')
20                ->addDefaultsIfNotSet()
21                ->children()
22                    ->scalarNode('name')
23                        ->isRequired()
24                        ->cannotBeEmpty()
25                        ->defaultValue('value')
26                    ->end()
27                ->end()
28            ->end()
29        ->end()
30    ;
```

Documenting the Option

All options can be documented using the *info()*[6] method.

Listing 14-21
```
1 $rootNode
2     ->children()
3         ->integerNode('entries_per_page')
4             ->info('This value is only used for the search results page.')
5             ->defaultValue(25)
6         ->end()
7     ->end()
8 ;
```

The info will be printed as a comment when dumping the configuration tree with the `config:dump-reference` command.

In YAML you may have:

Listing 14-22
```
1 # This value is only used for the search results page.
2 entries_per_page:     25
```

and in XML:

Listing 14-23
```
1 <!-- entries-per-page: This value is only used for the search results page. -->
2 <config entries-per-page="25" />
```

Optional Sections

If you have entire sections which are optional and can be enabled/disabled, you can take advantage of the shortcut *canBeEnabled()*[7] and *canBeDisabled()*[8] methods:

Listing 14-24

6. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/NodeDefinition.html#method_info
7. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#method_canBeEnabled
8. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#method_canBeDisabled

```
$arrayNode
    ->canBeEnabled()
;

// is equivalent to

$arrayNode
    ->treatFalseLike(array('enabled' => false))
    ->treatTrueLike(array('enabled' => true))
    ->treatNullLike(array('enabled' => true))
    ->children()
        ->booleanNode('enabled')
            ->defaultFalse()
;
```

The `canBeDisabled` method looks about the same except that the section would be enabled by default.

Merging Options

Extra options concerning the merge process may be provided. For arrays:

`performNoDeepMerging()`
: When the value is also defined in a second configuration array, don't try to merge an array, but overwrite it entirely

For all nodes:

`cannotBeOverwritten()`
: don't let other configuration arrays overwrite an existing value for this node

Appending Sections

If you have a complex configuration to validate then the tree can grow to be large and you may want to split it up into sections. You can do this by making a section a separate node and then appending it into the main tree with `append()`:

Listing 14-25

```
public function getConfigTreeBuilder()
{
    $treeBuilder = new TreeBuilder();
    $rootNode = $treeBuilder->root('database');

    $rootNode
        ->children()
            ->arrayNode('connection')
                ->children()
                    ->scalarNode('driver')
                        ->isRequired()
                        ->cannotBeEmpty()
                    ->end()
                    ->scalarNode('host')
                        ->defaultValue('localhost')
                    ->end()
                    ->scalarNode('username')->end()
                    ->scalarNode('password')->end()
                    ->booleanNode('memory')
```

```
20                          ->defaultFalse()
21                      ->end()
22                  ->end()
23                  ->append($this->addParametersNode())
24              ->end()
25          ->end()
26      ;
27
28      return $treeBuilder;
29  }
30
31  public function addParametersNode()
32  {
33      $builder = new TreeBuilder();
34      $node = $builder->root('parameters');
35
36      $node
37          ->isRequired()
38          ->requiresAtLeastOneElement()
39          ->useAttributeAsKey('name')
40          ->prototype('array')
41              ->children()
42                  ->scalarNode('value')->isRequired()->end()
43              ->end()
44          ->end()
45      ;
46
47      return $node;
48  }
```

This is also useful to help you avoid repeating yourself if you have sections of the config that are repeated in different places.

The example results in the following:

Listing 14-26

```
1  database:
2      connection:
3          driver:               ~ # Required
4          host:                 localhost
5          username:             ~
6          password:             ~
7          memory:               false
8          parameters:           # Required
9
10             # Prototype
11             name:
12                 value:                ~ # Required
```

Normalization

When the config files are processed they are first normalized, then merged and finally the tree is used to validate the resulting array. The normalization process is used to remove some of the differences that result from different configuration formats, mainly the differences between YAML and XML.

The separator used in keys is typically _ in YAML and - in XML. For example, `auto_connect` in YAML and `auto-connect` in XML. The normalization would make both of these `auto_connect`.

The target key will not be altered if it's mixed like `foo-bar_moo` or if it already exists.

Another difference between YAML and XML is in the way arrays of values may be represented. In YAML you may have:

Listing 14-27
```
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 14-28
```
<twig:config>
    <twig:extension>twig.extension.foo</twig:extension>
    <twig:extension>twig.extension.bar</twig:extension>
</twig:config>
```

This difference can be removed in normalization by pluralizing the key used in XML. You can specify that you want a key to be pluralized in this way with `fixXmlConfig()`:

Listing 14-29
```
$rootNode
    ->fixXmlConfig('extension')
    ->children()
        ->arrayNode('extensions')
            ->prototype('scalar')->end()
        ->end()
    ->end()
;
```

If it is an irregular pluralization you can specify the plural to use as a second argument:

Listing 14-30
```
$rootNode
    ->fixXmlConfig('child', 'children')
    ->children()
        ->arrayNode('children')
            // ...
        ->end()
    ->end()
;
```

As well as fixing this, `fixXmlConfig` ensures that single XML elements are still turned into an array. So you may have:

Listing 14-31
```
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 14-32
```
<connection>default</connection>
```

By default `connection` would be an array in the first case and a string in the second making it difficult to validate. You can ensure it is always an array with `fixXmlConfig`.

You can further control the normalization process if you need to. For example, you may want to allow a string to be set and used as a particular key or several keys to be set explicitly. So that, if everything apart from `name` is optional in this config:

Listing 14-33
```
connection:
    name:     my_mysql_connection
    host:     localhost
    driver:   mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 14-34
```
connection: my_mysql_connection
```

By changing a string value into an associative array with `name` as the key:

Listing 14-35
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
                ->ifString()
                ->then(function ($v) { return array('name' => $v); })
            ->end()
            ->children()
                ->scalarNode('name')->isRequired()
                // ...
            ->end()
        ->end()
    ->end()
;
```

Validation Rules

More advanced validation rules can be provided using the *ExprBuilder*[9]. This builder implements a fluent interface for a well-known control structure. The builder is used for adding advanced validation rules to node definitions, like:

Listing 14-36
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->validate()
                    ->ifNotInArray(array('mysql', 'sqlite', 'mssql'))
                        ->thenInvalid('Invalid database driver "%s"')
                    ->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

9. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/ExprBuilder.html

A validation rule always has an "if" part. You can specify this part in the following ways:

- `ifTrue()`
- `ifString()`
- `ifNull()`
- `ifArray()`
- `ifInArray()`
- `ifNotInArray()`
- `always()`

A validation rule also requires a "then" part:

- `then()`
- `thenEmptyArray()`
- `thenInvalid()`
- `thenUnset()`

Usually, "then" is a closure. Its return value will be used as a new value for the node, instead of the node's original value.

Processing Configuration Values

The *`Processor`*[10] uses the tree as it was built using the *`TreeBuilder`*[11] to process multiple arrays of configuration values that should be merged. If any value is not of the expected type, is mandatory and yet undefined, or could not be validated in some other way, an exception will be thrown. Otherwise the result is a clean array of configuration values:

Listing 14-37

```
1  use Symfony\Component\Yaml\Yaml;
2  use Symfony\Component\Config\Definition\Processor;
3  use Acme\DatabaseConfiguration;
4
5  $config1 = Yaml::parse(
6      file_get_contents(__DIR__.'/src/Matthias/config/config.yml')
7  );
8  $config2 = Yaml::parse(
9      file_get_contents(__DIR__.'/src/Matthias/config/config_extra.yml')
10 );
11
12 $configs = array($config1, $config2);
13
14 $processor = new Processor();
15 $configuration = new DatabaseConfiguration();
16 $processedConfiguration = $processor->processConfiguration(
17     $configuration,
18     $configs
19 );
```

10. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Processor.html
11. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html

Chapter 15
The Console Component

> The Console component eases the creation of beautiful and testable command line interfaces.

The Console component allows you to create command-line commands. Your console commands can be used for any recurring task, such as cronjobs, imports, or other batch jobs.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/console` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/console*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Creating a basic Command

To make a console command that greets you from the command line, create `GreetCommand.php` and add the following to it:

Listing 15-1

```
1  namespace Acme\Console\Command;
2
3  use Symfony\Component\Console\Command\Command;
4  use Symfony\Component\Console\Input\InputArgument;
5  use Symfony\Component\Console\Input\InputInterface;
6  use Symfony\Component\Console\Input\InputOption;
7  use Symfony\Component\Console\Output\OutputInterface;
8
```

1. `https://packagist.org/packages/symfony/console`

```
 9 class GreetCommand extends Command
10 {
11     protected function configure()
12     {
13         $this
14             ->setName('demo:greet')
15             ->setDescription('Greet someone')
16             ->addArgument(
17                 'name',
18                 InputArgument::OPTIONAL,
19                 'Who do you want to greet?'
20             )
21             ->addOption(
22                'yell',
23                null,
24                InputOption::VALUE_NONE,
25                'If set, the task will yell in uppercase letters'
26             )
27         ;
28     }
29
30     protected function execute(InputInterface $input, OutputInterface $output)
31     {
32         $name = $input->getArgument('name');
33         if ($name) {
34             $text = 'Hello '.$name;
35         } else {
36             $text = 'Hello';
37         }
38
39         if ($input->getOption('yell')) {
40             $text = strtoupper($text);
41         }
42
43         $output->writeln($text);
44     }
45 }
```

You also need to create the file to run at the command line which creates an `Application` and adds commands to it:

Listing 15-2

```
 1 #!/usr/bin/env php
 2 <?php
 3 // application.php
 4
 5 require __DIR__.'/vendor/autoload.php';
 6
 7 use Acme\Console\Command\GreetCommand;
 8 use Symfony\Component\Console\Application;
 9
10 $application = new Application();
11 $application->add(new GreetCommand());
12 $application->run();
```

Test the new console command by running the following

Listing 15-3

```
1 $ php application.php demo:greet Fabien
```

This will print the following to the command line:

Listing 15-4
```
1 Hello Fabien
```

You can also use the `--yell` option to make everything uppercase:

Listing 15-5
```
1 $ php application.php demo:greet Fabien --yell
```

This prints:

Listing 15-6
```
HELLO FABIEN
```

Command Lifecycle

Commands have three lifecycle methods:

***initialize()*[2] (optional)**
This method is executed before the `interact()` and the `execute()` methods. Its main purpose is to initialize variables used in the rest of the command methods.

***interact()*[3] (optional)**
This method is executed after `initialize()` and before `execute()`. Its purpose is to check if some of the options/arguments are missing and interactively ask the user for those values. This is the last place where you can ask for missing options/arguments. After this command, missing options/arguments will result in an error.

***execute()*[4] (required)**
This method is executed after `interact()` and `initialize()`. It contains the logic you want the command to execute.

Coloring the Output

By default, the Windows command console doesn't support output coloring. The Console component disables output coloring for Windows systems, but if your commands invoke other scripts which emit color sequences, they will be wrongly displayed as raw escape characters. Install the *ConEmu*[5], *ANSICON*[6] or *Mintty*[7] (used by default in GitBash and Cygwin) free applications to add coloring support to your Windows command console.

Whenever you output text, you can surround the text with tags to color its output. For example:

Listing 15-7
```
1 // green text
2 $output->writeln('<info>foo</info>');
3
4 // yellow text
5 $output->writeln('<comment>foo</comment>');
6
7 // black text on a cyan background
8 $output->writeln('<question>foo</question>');
```

2. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_initialize
3. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_interact
4. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_execute
5. https://conemu.github.io/
6. https://github.com/adoxa/ansicon/releases
7. https://mintty.github.io/

```
 9
10  // white text on a red background
11  $output->writeln('<error>foo</error>');
```

The closing tag can be replaced by </>, which revokes all formatting options established by the last opened tag.

It is possible to define your own styles using the class *OutputFormatterStyle*[8]:

Listing 15-8
```
1  use Symfony\Component\Console\Formatter\OutputFormatterStyle;
2
3  // ...
4  $style = new OutputFormatterStyle('red', 'yellow', array('bold', 'blink'));
5  $output->getFormatter()->setStyle('fire', $style);
6  $output->writeln('<fire>foo</>');
```

Available foreground and background colors are: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`.

And available options are: `bold`, `underscore`, `blink`, `reverse` (enables the "reverse video" mode where the background and foreground colors are swapped) and `conceal` (sets the foreground color to transparent, making the typed text invisible - although it can be selected and copied; this option is commonly used when asking the user to type sensitive information).

You can also set these colors and options inside the tagname:

Listing 15-9
```
1  // green text
2  $output->writeln('<fg=green>foo</>');
3
4  // black text on a cyan background
5  $output->writeln('<fg=black;bg=cyan>foo</>');
6
7  // bold text on a yellow background
8  $output->writeln('<bg=yellow;options=bold>foo</>');
```

Verbosity Levels

The console has five verbosity levels. These are defined in the *OutputInterface*[9]:

Value	Meaning	Console option
`OutputInterface::VERBOSITY_QUIET`	Do not output any messages	`-q` or `--quiet`
`OutputInterface::VERBOSITY_NORMAL`	The default verbosity level	(none)
`OutputInterface::VERBOSITY_VERBOSE`	Increased verbosity of messages	`-v`
`OutputInterface::VERBOSITY_VERY_VERBOSE`	Informative non essential messages	`-vv`
`OutputInterface::VERBOSITY_DEBUG`	Debug messages	`-vvv`

8. http://api.symfony.com/3.0/Symfony/Component/Console/Formatter/OutputFormatterStyle.html
9. http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html

The full exception stacktrace is printed if the VERBOSITY_VERBOSE level or above is used.

It is possible to print a message in a command for only a specific verbosity level. For example:

Listing 15-10
```
if ($output->getVerbosity() >= OutputInterface::VERBOSITY_VERBOSE) {
    $output->writeln(...);
}
```

There are also more semantic methods you can use to test for each of the verbosity levels:

Listing 15-11
```
if ($output->isQuiet()) {
    // ...
}

if ($output->isVerbose()) {
    // ...
}

if ($output->isVeryVerbose()) {
    // ...
}

if ($output->isDebug()) {
    // ...
}
```

These semantic methods are defined in the `OutputInterface` starting from Symfony 3.0. In previous Symfony versions they are defined in the different implementations of the interface (e.g. *Output*[10]) in order to keep backwards compatibility.

When the quiet level is used, all output is suppressed as the default *write()*[11] method returns without actually printing.

The MonologBridge provides a *ConsoleHandler*[12] class that allows you to display messages on the console. This is cleaner than wrapping your output calls in conditions. For an example use in the Symfony Framework, see *How to Configure Monolog to Display Console Messages*.

Using Command Arguments

The most interesting part of the commands are the arguments and options that you can make available. Arguments are the strings - separated by spaces - that come after the command name itself. They are ordered, and can be optional or required. For example, add an optional `last_name` argument to the command and make the `name` argument required:

Listing 15-12

10. http://api.symfony.com/3.0/Symfony/Component/Console/Output/Output.html
11. http://api.symfony.com/3.0/Symfony/Component/Console/Output/Output.html#method_write
12. http://api.symfony.com/3.0/Symfony/Bridge/Monolog/Handler/ConsoleHandler.html

```
1 $this
2     // ...
3     ->addArgument(
4         'name',
5         InputArgument::REQUIRED,
6         'Who do you want to greet?'
7     )
8     ->addArgument(
9         'last_name',
10        InputArgument::OPTIONAL,
11        'Your last name?'
12    );
```

You now have access to a `last_name` argument in your command:

Listing 15-13
```
if ($lastName = $input->getArgument('last_name')) {
    $text .= ' '.$lastName;
}
```

The command can now be used in either of the following ways:

Listing 15-14
```
1 $ php application.php demo:greet Fabien
2 $ php application.php demo:greet Fabien Potencier
```

It is also possible to let an argument take a list of values (imagine you want to greet all your friends). For this it must be specified at the end of the argument list:

Listing 15-15
```
1 $this
2     // ...
3     ->addArgument(
4         'names',
5         InputArgument::IS_ARRAY,
6         'Who do you want to greet (separate multiple names with a space)?'
7     );
```

To use this, just specify as many names as you want:

Listing 15-16
```
1 $ php application.php demo:greet Fabien Ryan Bernhard
```

You can access the `names` argument as an array:

Listing 15-17
```
if ($names = $input->getArgument('names')) {
    $text .= ' '.implode(', ', $names);
}
```

There are three argument variants you can use:

Mode	**Value**
`InputArgument::REQUIRED`	The argument is required
`InputArgument::OPTIONAL`	The argument is optional and therefore can be omitted
`InputArgument::IS_ARRAY`	The argument can contain an indefinite number of arguments and must be used at the end of the argument list

You can combine `IS_ARRAY` with `REQUIRED` and `OPTIONAL` like this:

Listing 15-18
```
$this
    // ...
    ->addArgument(
        'names',
        InputArgument::IS_ARRAY | InputArgument::REQUIRED,
        'Who do you want to greet (separate multiple names with a space)?'
    );
```

Using Command Options

Unlike arguments, options are not ordered (meaning you can specify them in any order) and are specified with two dashes (e.g. `--yell` - you can also declare a one-letter shortcut that you can call with a single dash like `-y`). Options are *always* optional, and can be setup to accept a value (e.g. `--dir=src`) or simply as a boolean flag without a value (e.g. `--yell`).

There is nothing forbidding you to create a command with an option that optionally accepts a value. However, there is no way you can distinguish when the option was used without a value (`command --yell`) or when it wasn't used at all (`command`). In both cases, the value retrieved for the option will be `null`.

For example, add a new option to the command that can be used to specify how many times in a row the message should be printed:

Listing 15-19
```
$this
    // ...
    ->addOption(
        'iterations',
        null,
        InputOption::VALUE_REQUIRED,
        'How many times should the message be printed?',
        1
    );
```

Next, use this in the command to print the message multiple times:

Listing 15-20
```
for ($i = 0; $i < $input->getOption('iterations'); $i++) {
    $output->writeln($text);
}
```

Now, when you run the task, you can optionally specify a `--iterations` flag:

Listing 15-21
```
$ php application.php demo:greet Fabien
$ php application.php demo:greet Fabien --iterations=5
```

The first example will only print once, since `iterations` is empty and defaults to `1` (the last argument of `addOption`). The second example will print five times.

Recall that options don't care about their order. So, either of the following will work:

Listing 15-22
```
$ php application.php demo:greet Fabien --iterations=5 --yell
$ php application.php demo:greet Fabien --yell --iterations=5
```

There are 4 option variants you can use:

Option	Value
`InputOption::VALUE_IS_ARRAY`	This option accepts multiple values (e.g. `--dir=/foo --dir=/bar`)
`InputOption::VALUE_NONE`	Do not accept input for this option (e.g. `--yell`)
`InputOption::VALUE_REQUIRED`	This value is required (e.g. `--iterations=5`), the option itself is still optional
`InputOption::VALUE_OPTIONAL`	This option may or may not have a value (e.g. `--yell` or `--yell=loud`)

You can combine `VALUE_IS_ARRAY` with `VALUE_REQUIRED` or `VALUE_OPTIONAL` like this:

Listing 15-23
```
$this
    // ...
    ->addOption(
        'colors',
        null,
        InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY,
        'Which colors do you like?',
        array('blue', 'red')
    );
```

Console Helpers

The console component also contains a set of "helpers" - different small tools capable of helping you with different tasks:

- *Question Helper*: interactively ask the user for information
- *Formatter Helper*: customize the output colorization
- *Progress Bar*: shows a progress bar
- *Table*: displays tabular data as a table

Testing Commands

Symfony provides several tools to help you test your commands. The most useful one is the *CommandTester*[13] class. It uses special input and output classes to ease testing without a real console:

Listing 15-24
```
use Acme\Console\Command\GreetCommand;
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        $application = new Application();
        $application->add(new GreetCommand());
```

13. http://api.symfony.com/3.0/Symfony/Component/Console/Tester/CommandTester.html

```
11
12         $command = $application->find('demo:greet');
13         $commandTester = new CommandTester($command);
14         $commandTester->execute(array('command' => $command->getName()));
15
16         $this->assertRegExp('/.../', $commandTester->getDisplay());
17
18         // ...
19     }
20 }
```

The *getDisplay()*[14] method returns what would have been displayed during a normal call from the console.

You can test sending arguments and options to the command by passing them as an array to the *execute()*[15] method:

Listing 15-25

```
1  use Acme\Console\Command\GreetCommand;
2  use Symfony\Component\Console\Application;
3  use Symfony\Component\Console\Tester\CommandTester;
4
5  class ListCommandTest extends \PHPUnit_Framework_TestCase
6  {
7      // ...
8
9      public function testNameIsOutput()
10     {
11         $application = new Application();
12         $application->add(new GreetCommand());
13
14         $command = $application->find('demo:greet');
15         $commandTester = new CommandTester($command);
16         $commandTester->execute(array(
17             'command'      => $command->getName(),
18             'name'         => 'Fabien',
19             '--iterations' => 5,
20         ));
21
22         $this->assertRegExp('/Fabien/', $commandTester->getDisplay());
23     }
24 }
```

You can also test a whole console application by using *ApplicationTester*[16].

Calling an Existing Command

If a command depends on another one being run before it, instead of asking the user to remember the order of execution, you can call it directly yourself. This is also useful if you want to create a "meta" command that just runs a bunch of other commands (for instance, all commands that need to be run

14. http://api.symfony.com/3.0/Symfony/Component/Console/Tester/CommandTester.html#method_getDisplay
15. http://api.symfony.com/3.0/Symfony/Component/Console/Tester/CommandTester.html#method_execute
16. http://api.symfony.com/3.0/Symfony/Component/Console/Tester/ApplicationTester.html

when the project's code has changed on the production servers: clearing the cache, generating Doctrine2 proxies, dumping Assetic assets, ...).

Calling a command from another one is straightforward:

Listing 15-26
```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $command = $this->getApplication()->find('demo:greet');

    $arguments = array(
        'command' => 'demo:greet',
        'name'    => 'Fabien',
        '--yell'  => true,
    );

    $greetInput = new ArrayInput($arguments);
    $returnCode = $command->run($greetInput, $output);

    // ...
}
```

First, you *find()*[17] the command you want to execute by passing the command name. Then, you need to create a new *ArrayInput*[18] with the arguments and options you want to pass to the command.

Eventually, calling the `run()` method actually executes the command and returns the returned code from the command (return value from command's `execute()` method).

If you want to suppress the output of the executed command, pass a *NullOutput*[19] as the second argument to `$command->execute()`.

Note that all the commands will run in the same process and some of Symfony's built-in commands may not work well this way. For instance, the `cache:clear` and `cache:warmup` commands change some class definitions, so running something after them is likely to break.

Most of the time, calling a command from code that is not executed on the command line is not a good idea for several reasons. First, the command's output is optimized for the console. But more important, you can think of a command as being like a controller; it should use the model to do something and display feedback to the user. So, instead of calling a command from the Web, refactor your code and move the logic to a new class.

Learn More!

- *Using Console Commands, Shortcuts and Built-in Commands*
- *Building a single Command Application*
- *Changing the Default Command*
- *Using Events*
- *Understanding how Console Arguments Are Handled*

17. http://api.symfony.com/3.0/Symfony/Component/Console/Application.html#method_find
18. http://api.symfony.com/3.0/Symfony/Component/Console/Input/ArrayInput.html
19. http://api.symfony.com/3.0/Symfony/Component/Console/Output/NullOutput.html

Chapter 16
Using Console Commands, Shortcuts and Built-in Commands

In addition to the options you specify for your commands, there are some built-in options as well as a couple of built-in commands for the Console component.

These examples assume you have added a file `application.php` to run at the cli:

Listing 16-1
```
1  #!/usr/bin/env php
2  <?php
3  // application.php
4
5  use Symfony\Component\Console\Application;
6
7  $application = new Application();
8  // ...
9  $application->run();
```

Built-in Commands

There is a built-in command `list` which outputs all the standard options and the registered commands:

Listing 16-2
```
1  $ php application.php list
```

You can get the same output by not running any command as well

Listing 16-3
```
1  $ php application.php
```

The help command lists the help information for the specified command. For example, to get the help for the `list` command:

Listing 16-4
```
1 $ php application.php help list
```

Running `help` without specifying a command will list the global options:

Listing 16-5
```
1 $ php application.php help
```

Global Options

You can get help information for any command with the `--help` option. To get help for the list command:

Listing 16-6
```
1 $ php application.php list --help
2 $ php application.php list -h
```

You can suppress output with:

Listing 16-7
```
1 $ php application.php list --quiet
2 $ php application.php list -q
```

You can get more verbose messages (if this is supported for a command) with:

Listing 16-8
```
1 $ php application.php list --verbose
2 $ php application.php list -v
```

To output even more verbose messages you can use these options:

Listing 16-9
```
1 $ php application.php list -vv
2 $ php application.php list -vvv
```

If you set the optional arguments to give your application a name and version:

Listing 16-10
```
$application = new Application('Acme Console Application', '1.2');
```

then you can use:

Listing 16-11
```
1 $ php application.php list --version
2 $ php application.php list -V
```

to get this information output:

Listing 16-12
```
1 Acme Console Application version 1.2
```

If you do not provide both arguments then it will just output:

Listing 16-13
```
1 console tool
```

You can force turning on ANSI output coloring with:

Listing 16-14
```
1 $ php application.php list --ansi
```

or turn it off with:

Listing 16-15
```
1 $ php application.php list --no-ansi
```

You can suppress any interactive questions from the command you are running with:

Listing 16-16
```
1 $ php application.php list --no-interaction
2 $ php application.php list -n
```

Shortcut Syntax

You do not have to type out the full command names. You can just type the shortest unambiguous name to run a command. So if there are non-clashing commands, then you can run `help` like this:

Listing 16-17
```
1 $ php application.php h
```

If you have commands using `:` to namespace commands then you just have to type the shortest unambiguous text for each part. If you have created the `demo:greet` as shown in *The Console Component* then you can run it with:

Listing 16-18
```
1 $ php application.php d:g Fabien
```

If you enter a short command that's ambiguous (i.e. there are more than one command that match), then no command will be run and some suggestions of the possible commands to choose from will be output.

Chapter 17
Changing the Default Command

The Console component will always run the `ListCommand` when no command name is passed. In order to change the default command you just need to pass the command name to the `setDefaultCommand` method:

Listing 17-1
```
namespace Acme\Console\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class HelloWorldCommand extends Command
{
    protected function configure()
    {
        $this->setName('hello:world')
            ->setDescription('Outputs \'Hello World\'');
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $output->writeln('Hello World');
    }
}
```

Executing the application and changing the default Command:

Listing 17-2
```
// application.php

use Acme\Console\Command\HelloWorldCommand;
use Symfony\Component\Console\Application;

$command = new HelloWorldCommand();
$application = new Application();
$application->add($command);
```

```
9 $application->setDefaultCommand($command->getName());
10 $application->run();
```

Test the new default console command by running the following:

Listing 17-3
```
1 $ php application.php
```

This will print the following to the command line:

Listing 17-4
```
1 Hello World
```

This feature has a limitation: you cannot use it with any Command arguments.

Learn More!

- *Building a single Command Application*

Chapter 18
Building a single Command Application

When building a command line tool, you may not need to provide several commands. In such case, having to pass the command name each time is tedious. Fortunately, it is possible to remove this need by extending the application:

Listing 18-1

```php
namespace Acme\Tool;

use Symfony\Component\Console\Application;
use Symfony\Component\Console\Input\InputInterface;

class MyApplication extends Application
{
    /**
     * Gets the name of the command based on input.
     *
     * @param InputInterface $input The input interface
     *
     * @return string The command name
     */
    protected function getCommandName(InputInterface $input)
    {
        // This should return the name of your command.
        return 'my_command';
    }

    /**
     * Gets the default commands that should always be available.
     *
     * @return array An array of default Command instances
     */
    protected function getDefaultCommands()
    {
        // Keep the core default commands to have the HelpCommand
        // which is used when using the --help option
        $defaultCommands = parent::getDefaultCommands();

        $defaultCommands[] = new MyCommand();
```

```
33
34             return $defaultCommands;
35         }
36
37         /**
38          * Overridden so that the application doesn't expect the command
39          * name to be the first argument.
40          */
41         public function getDefinition()
42         {
43             $inputDefinition = parent::getDefinition();
44             // clear out the normal first argument, which is the command name
45             $inputDefinition->setArguments();
46
47             return $inputDefinition;
48         }
49     }
```

When calling your console script, the command `MyCommand` will then always be used, without having to pass its name.

You can also simplify how you execute the application:

Listing 18-2

```
1  #!/usr/bin/env php
2  <?php
3  // command.php
4
5  use Acme\Tool\MyApplication;
6
7  $application = new MyApplication();
8  $application->run();
```

Chapter 19

Understanding how Console Arguments Are Handled

It can be difficult to understand the way arguments are handled by the console application. The Symfony Console application, like many other CLI utility tools, follows the behavior described in the *docopt*[1] standards.

Have a look at the following command that has three options:

Listing 19-1

```
namespace Acme\Console\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputDefinition;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class DemoArgsCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('demo:args')
            ->setDescription('Describe args behaviors')
            ->setDefinition(
                new InputDefinition(array(
                    new InputOption('foo', 'f'),
                    new InputOption('bar', 'b', InputOption::VALUE_REQUIRED),
                    new InputOption('cat', 'c', InputOption::VALUE_OPTIONAL),
                ))
            );
    }

```

1. http://docopt.org/

```
26     protected function execute(InputInterface $input, OutputInterface $output)
27     {
28         // ...
29     }
30 }
```

Since the `foo` option doesn't accept a value, it will be either `false` (when it is not passed to the command) or `true` (when `--foo` was passed by the user). The value of the `bar` option (and its `b` shortcut respectively) is required. It can be separated from the option name either by spaces or `=` characters. The `cat` option (and its `c` shortcut) behaves similar except that it doesn't require a value. Have a look at the following table to get an overview of the possible ways to pass options:

Input	foo	bar	cat
`--bar=Hello`	`false`	`"Hello"`	`null`
`--bar Hello`	`false`	`"Hello"`	`null`
`-b=Hello`	`false`	`"=Hello"`	`null`
`-b Hello`	`false`	`"Hello"`	`null`
`-bHello`	`false`	`"Hello"`	`null`
`-fcWorld -b Hello`	`true`	`"Hello"`	`"World"`
`-cfWorld -b Hello`	`false`	`"Hello"`	`"fWorld"`
`-cbWorld`	`false`	`null`	`"bWorld"`

Things get a little bit more tricky when the command also accepts an optional argument:

Listing 19-2
```
1 // ...
2
3 new InputDefinition(array(
4     // ...
5     new InputArgument('arg', InputArgument::OPTIONAL),
6 ));
```

You might have to use the special `--` separator to separate options from arguments. Have a look at the fifth example in the following table where it is used to tell the command that `World` is the value for `arg` and not the value of the optional `cat` option:

Input	bar	cat	arg
`--bar Hello`	`"Hello"`	`null`	`null`
`--bar Hello World`	`"Hello"`	`null`	`"World"`
`--bar "Hello World"`	`"Hello World"`	`null`	`null`
`--bar Hello --cat World`	`"Hello"`	`"World"`	`null`
`--bar Hello --cat -- World`	`"Hello"`	`null`	`"World"`
`-b Hello -c World`	`"Hello"`	`"World"`	`null`

Chapter 20

Using Events

The Application class of the Console component allows you to optionally hook into the lifecycle of a console application via events. Instead of reinventing the wheel, it uses the Symfony EventDispatcher component to do the work:

Listing 20-1

```
use Symfony\Component\Console\Application;
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();

$application = new Application();
$application->setDispatcher($dispatcher);
$application->run();
```

Console events are only triggered by the main command being executed. Commands called by the main command will not trigger any event.

The `ConsoleEvents::COMMAND` Event

Typical Purposes: Doing something before any command is run (like logging which command is going to be executed), or displaying something about the event to be executed.

Just before executing any command, the `ConsoleEvents::COMMAND` event is dispatched. Listeners receive a *ConsoleCommandEvent*[1] event:

Listing 20-2

```
use Symfony\Component\Console\Event\ConsoleCommandEvent;
use Symfony\Component\Console\ConsoleEvents;

$dispatcher->addListener(ConsoleEvents::COMMAND, function (ConsoleCommandEvent $event) {
```

1. http://api.symfony.com/3.0/Symfony/Component/Console/Event/ConsoleCommandEvent.html

```
5     // get the input instance
6     $input = $event->getInput();
7
8     // get the output instance
9     $output = $event->getOutput();
10
11    // get the command to be executed
12    $command = $event->getCommand();
13
14    // write something about the command
15    $output->writeln(sprintf('Before running command <info>%s</info>',
16 $command->getName()));
17
18    // get the application
19    $application = $command->getApplication();
   });
```

Disable Commands inside Listeners

Using the *disableCommand()*[2] method, you can disable a command inside a listener. The application will then *not* execute the command, but instead will return the code `113` (defined in `ConsoleCommandEvent::RETURN_CODE_DISABLED`). This code is one of the *reserved exit codes*[3] for console commands that conform with the C/C++ standard.:

Listing 20-3

```
1  use Symfony\Component\Console\Event\ConsoleCommandEvent;
2  use Symfony\Component\Console\ConsoleEvents;
3
4  $dispatcher->addListener(ConsoleEvents::COMMAND, function (ConsoleCommandEvent $event) {
5      // get the command to be executed
6      $command = $event->getCommand();
7
8      // ... check if the command can be executed
9
10     // disable the command, this will result in the command being skipped
11     // and code 113 being returned from the Application
12     $event->disableCommand();
13
14     // it is possible to enable the command in a later listener
15     if (!$event->commandShouldRun()) {
16         $event->enableCommand();
17     }
18 });
```

The `ConsoleEvents::TERMINATE` Event

Typical Purposes: To perform some cleanup actions after the command has been executed.

After the command has been executed, the `ConsoleEvents::TERMINATE` event is dispatched. It can be used to do any actions that need to be executed for all commands or to cleanup what you initiated in a `ConsoleEvents::COMMAND` listener (like sending logs, closing a database connection, sending emails, ...). A listener might also change the exit code.

2. http://api.symfony.com/3.0/Symfony/Component/Console/Event/ConsoleCommandEvent.html#method_disableCommand
3. http://www.tldp.org/LDP/abs/html/exitcodes.html

Listeners receive a *ConsoleTerminateEvent*[4] event:

Listing 20-4

```
use Symfony\Component\Console\Event\ConsoleTerminateEvent;
use Symfony\Component\Console\ConsoleEvents;

$dispatcher->addListener(ConsoleEvents::TERMINATE, function (ConsoleTerminateEvent $event)
{
    // get the output
    $output = $event->getOutput();

    // get the command that has been executed
    $command = $event->getCommand();

    // display something
    $output->writeln(sprintf('After running command <info>%s</info>',
$command->getName()));

    // change the exit code
    $event->setExitCode(128);
});
```

This event is also dispatched when an exception is thrown by the command. It is then dispatched just before the `ConsoleEvents::EXCEPTION` event. The exit code received in this case is the exception code.

The `ConsoleEvents::EXCEPTION` Event

Typical Purposes: Handle exceptions thrown during the execution of a command.

Whenever an exception is thrown by a command, the `ConsoleEvents::EXCEPTION` event is dispatched. A listener can wrap or change the exception or do anything useful before the exception is thrown by the application.

Listeners receive a *ConsoleExceptionEvent*[5] event:

Listing 20-5

```
use Symfony\Component\Console\Event\ConsoleExceptionEvent;
use Symfony\Component\Console\ConsoleEvents;

$dispatcher->addListener(ConsoleEvents::EXCEPTION, function (ConsoleExceptionEvent $event)
{
    $output = $event->getOutput();

    $command = $event->getCommand();

    $output->writeln(sprintf('Oops, exception thrown while running command
<info>%s</info>', $command->getName()));

    // get the current exit code (the exception code or the exit code set by a
ConsoleEvents::TERMINATE event)
    $exitCode = $event->getExitCode();

    // change the exception to another one
    $event->setException(new \LogicException('Caught exception', $exitCode,
```

4. http://api.symfony.com/3.0/Symfony/Component/Console/Event/ConsoleTerminateEvent.html
5. http://api.symfony.com/3.0/Symfony/Component/Console/Event/ConsoleExceptionEvent.html

```
    $event->getException()));
});
```

Chapter 21
Using the Logger

The Console component comes with a standalone logger complying with the *PSR-3*[1] standard. Depending on the verbosity setting, log messages will be sent to the *`OutputInterface`*[2] instance passed as a parameter to the constructor.

The logger does not have any external dependency except `php-fig/log`. This is useful for console applications and commands needing a lightweight PSR-3 compliant logger:

Listing 21-1

```
namespace Acme;

use Psr\Log\LoggerInterface;

class MyDependency
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function doStuff()
    {
        $this->logger->info('I love Tony Vairelles\' hairdresser.');
    }
}
```

You can rely on the logger to use this dependency inside a command:

Listing 21-2

```
namespace Acme\Console\Command;

use Acme\MyDependency;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
```

1. http://www.php-fig.org/psr/psr-3/
2. http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html

```php
 6  use Symfony\Component\Console\Output\OutputInterface;
 7  use Symfony\Component\Console\Logger\ConsoleLogger;
 8
 9  class MyCommand extends Command
10  {
11      protected function configure()
12      {
13          $this
14              ->setName('my:command')
15              ->setDescription(
16                  'Use an external dependency requiring a PSR-3 logger'
17              )
18          ;
19      }
20
21      protected function execute(InputInterface $input, OutputInterface $output)
22      {
23          $logger = new ConsoleLogger($output);
24
25          $myDependency = new MyDependency($logger);
26          $myDependency->doStuff();
27      }
28  }
```

The dependency will use the instance of `ConsoleLogger`[3] as logger. Log messages emitted will be displayed on the console output.

Verbosity

Depending on the verbosity level that the command is run, messages may or may not be sent to the `OutputInterface`[4] instance.

By default, the console logger behaves like the *Monolog's Console Handler*. The association between the log level and the verbosity can be configured through the second parameter of the `ConsoleLogger`[5] constructor:

Listing 21-3

```php
1  use Psr\Log\LogLevel;
2  // ...
3
4  $verbosityLevelMap = array(
5      LogLevel::NOTICE => OutputInterface::VERBOSITY_NORMAL,
6      LogLevel::INFO   => OutputInterface::VERBOSITY_NORMAL,
7  );
8  $logger = new ConsoleLogger($output, $verbosityLevelMap);
```

Color

The logger outputs the log messages formatted with a color reflecting their level. This behavior is configurable through the third parameter of the constructor:

Listing 21-4

3. http://api.symfony.com/3.0/Symfony/Component/Console/Logger/ConsoleLogger.html
4. http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html
5. http://api.symfony.com/3.0/Symfony/Component/Console/ConsoleLogger.html

```php
// ...
$formatLevelMap = array(
    LogLevel::CRITICAL => ConsoleLogger::INFO,
    LogLevel::DEBUG    => ConsoleLogger::ERROR,
);
$logger = new ConsoleLogger($output, array(), $formatLevelMap);
```

Chapter 22

Dialog Helper

The Dialog Helper was deprecated in Symfony 2.5 and removed in Symfony 3.0. You should now use the *Question Helper* instead, which is simpler to use.

Chapter 23
Formatter Helper

The Formatter helpers provides functions to format the output with colors. You can do more advanced things with this helper than you can in Coloring the Output.

The *FormatterHelper*[1] is included in the default helper set, which you can get by calling *getHelperSet()*[2]:

Listing 23-1
```
$formatter = $this->getHelper('formatter');
```

The methods return a string, which you'll usually render to the console by passing it to the *OutputInterface::writeln*[3] method.

Print Messages in a Section

Symfony offers a defined style when printing a message that belongs to some "section". It prints the section in color and with brackets around it and the actual message to the right of this. Minus the color, it looks like this:

Listing 23-2
```
1 [SomeSection] Here is some message related to that section
```

To reproduce this style, you can use the *formatSection()*[4] method:

Listing 23-3
```
1 $formattedLine = $formatter->formatSection(
2     'SomeSection',
3     'Here is some message related to that section'
4 );
5 $output->writeln($formattedLine);
```

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/FormatterHelper.html
2. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_getHelperSet
3. http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html#method_writeln
4. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/FormatterHelper.html#method_formatSection

Print Messages in a Block

Sometimes you want to be able to print a whole block of text with a background color. Symfony uses this when printing error messages.

If you print your error message on more than one line manually, you will notice that the background is only as long as each individual line. Use the *formatBlock()*[5] to generate a block output:

Listing 23-4
```
$errorMessages = array('Error!', 'Something went wrong');
$formattedBlock = $formatter->formatBlock($errorMessages, 'error');
$output->writeln($formattedBlock);
```

As you can see, passing an array of messages to the *formatBlock()*[6] method creates the desired output. If you pass `true` as third parameter, the block will be formatted with more padding (one blank line above and below the messages and 2 spaces on the left and right).

The exact "style" you use in the block is up to you. In this case, you're using the pre-defined `error` style, but there are other styles, or you can create your own. See Coloring the Output.

5. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/FormatterHelper.html#method_formatBlock
6. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/FormatterHelper.html#method_formatBlock

Chapter 24
Process Helper

The Process Helper shows processes as they're running and reports useful information about process status.

To display process details, use the *ProcessHelper*[1] and run your command with verbosity. For example, running the following code with a very verbose verbosity (e.g. -vv):

Listing 24-1
```
use Symfony\Component\Process\ProcessBuilder;

$helper = $this->getHelper('process');
$process = ProcessBuilder::create(array('figlet', 'Symfony'))->getProcess();

$helper->run($output, $process);
```

will result in this output:

```
RUN  figlet Symfony
RES  Command ran successfully
```

It will result in more detailed output with debug verbosity (e.g. `-vvv`):

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProcessHelper.html

In case the process fails, debugging is easier:

```
RUN  foo
ERR  sh: foo: command not found
ERR
RES  127 Command did not run successfully
```

Arguments

There are three ways to use the process helper:

- Using a command line string:

Listing 24-2
```
// ...
$helper->run($output, 'figlet Symfony');
```

- An array of arguments:

Listing 24-3
```
// ...
$helper->run($output, array('figlet', 'Symfony'));
```

When running the helper against an array of arguments, be aware that these will be automatically escaped.

- Passing a `Process`[2] instance:

Listing 24-4
```
1 use Symfony\Component\Process\ProcessBuilder;
2
3 // ...
4 $process = ProcessBuilder::create(array('figlet', 'Symfony'))->getProcess();
5
6 $helper->run($output, $process);
```

2. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html

Customized Display

You can display a customized error message using the third argument of the *run()*[3] method:

Listing 24-5
```
$helper->run($output, $process, 'The process failed :(');
```

A custom process callback can be passed as the fourth argument. Refer to the *Process Component* for callback documentation:

Listing 24-6
```
1 use Symfony\Component\Process\Process;
2
3 $helper->run($output, $process, 'The process failed :(', function ($type, $data) {
4     if (Process::ERR === $type) {
5         // ... do something with the stderr output
6     } else {
7         // ... do something with the stdout
8     }
9 });
```

3. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProcessHelper.html#method_run

Chapter 25

Progress Bar

When executing longer-running commands, it may be helpful to show progress information, which updates as your command runs:

```
Starting the demo... fingers crossed
 1/15                               6%
 1 sec                          3.5 MB
```

To display progress details, use the *ProgressBar*[1], pass it a total number of units, and advance the progress as the command executes:

Listing 25-1

```php
use Symfony\Component\Console\Helper\ProgressBar;

// create a new progress bar (50 units)
$progress = new ProgressBar($output, 50);

// start and displays the progress bar
$progress->start();

$i = 0;
while ($i++ < 50) {
    // ... do some work

    // advance the progress bar 1 unit
    $progress->advance();

    // you can also advance the progress bar by more than 1 unit
    // $progress->advance(3);
}

```

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html

```
20 // ensure that the progress bar is at 100%
21 $progress->finish();
```

Instead of advancing the bar by a number of steps (with the *advance()*[2] method), you can also set the current progress by calling the *setProgress()*[3] method.

If your platform doesn't support ANSI codes, updates to the progress bar are added as new lines. To prevent the output from being flooded, adjust the *setRedrawFrequency()*[4] accordingly. By default, when using a `max`, the redraw frequency is set to *10%* of your `max`.

If you don't know the number of steps in advance, just omit the steps argument when creating the *ProgressBar*[5] instance:

Listing 25-2

```
$progress = new ProgressBar($output);
```

The progress will then be displayed as a throbber:

Listing 25-3

```
1 # no max steps (displays it like a throbber)
2     0 [>---------------------------]
3     5 [----->----------------------]
4     5 [============================]
5
6 # max steps defined
7  0/3 [>---------------------------]   0%
8  1/3 [=========>------------------]  33%
9  3/3 [============================] 100%
```

Whenever your task is finished, don't forget to call *finish()*[6] to ensure that the progress bar display is refreshed with a 100% completion.

If you want to output something while the progress bar is running, call *clear()*[7] first. After you're done, call *display()*[8] to show the progress bar again.

Customizing the Progress Bar

Built-in Formats

By default, the information rendered on a progress bar depends on the current level of verbosity of the `OutputInterface` instance:

Listing 25-4

```
1 # OutputInterface::VERBOSITY_NORMAL (CLI with no verbosity flag)
2  0/3 [>---------------------------]   0%
3  1/3 [=========>------------------]  33%
```

2. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_advance
3. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_setProgress
4. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_setRedrawFrequency
5. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html
6. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_finish
7. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_clear
8. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_display

```
 3/3 [============================] 100%

# OutputInterface::VERBOSITY_VERBOSE (-v)
 0/3 [>---------------------------]   0%  1 sec
 1/3 [=========>------------------]  33%  1 sec
 3/3 [============================] 100%  1 sec

# OutputInterface::VERBOSITY_VERY_VERBOSE (-vv)
 0/3 [>---------------------------]   0%  1 sec
 1/3 [=========>------------------]  33%  1 sec
 3/3 [============================] 100%  1 sec

# OutputInterface::VERBOSITY_DEBUG (-vvv)
 0/3 [>---------------------------]   0%  1 sec/1 sec  1.0 MB
 1/3 [=========>------------------]  33%  1 sec/1 sec  1.0 MB
 3/3 [============================] 100%  1 sec/1 sec  1.0 MB
```

If you call a command with the quiet flag (`-q`), the progress bar won't be displayed.

Instead of relying on the verbosity mode of the current command, you can also force a format via `setFormat()`:

Listing 25-5
```
$bar->setFormat('verbose');
```

The built-in formats are the following:

- `normal`
- `verbose`
- `very_verbose`
- `debug`

If you don't set the number of steps for your progress bar, use the `_nomax` variants:

- `normal_nomax`
- `verbose_nomax`
- `very_verbose_nomax`
- `debug_nomax`

Custom Formats

Instead of using the built-in formats, you can also set your own:

Listing 25-6
```
$bar->setFormat('%bar%');
```

This sets the format to only display the progress bar itself:

Listing 25-7
```
>---------------------------
=========>------------------
============================
```

A progress bar format is a string that contains specific placeholders (a name enclosed with the `%` character); the placeholders are replaced based on the current progress of the bar. Here is a list of the built-in placeholders:

- `current`: The current step;
- `max`: The maximum number of steps (or 0 if no max is defined);
- `bar`: The bar itself;
- `percent`: The percentage of completion (not available if no max is defined);
- `elapsed`: The time elapsed since the start of the progress bar;
- `remaining`: The remaining time to complete the task (not available if no max is defined);
- `estimated`: The estimated time to complete the task (not available if no max is defined);
- `memory`: The current memory usage;
- `message`: The current message attached to the progress bar.

For instance, here is how you could set the format to be the same as the **debug** one:

Listing 25-8
```
$bar->setFormat(' %current%/%max% [%bar%] %percent:3s%% %elapsed:6s%/%estimated:-6s%
%memory:6s%');
```

Notice the `:6s` part added to some placeholders? That's how you can tweak the appearance of the bar (formatting and alignment). The part after the colon (`:`) is used to set the `sprintf` format of the string.

The `message` placeholder is a bit special as you must set the value yourself:

Listing 25-9
```
1  $bar->setMessage('Task starts');
2  $bar->start();
3
4  $bar->setMessage('Task in progress...');
5  $bar->advance();
6
7  // ...
8
9  $bar->setMessage('Task is finished');
10 $bar->finish();
```

Instead of setting the format for a given instance of a progress bar, you can also define global formats:

Listing 25-10
```
ProgressBar::setFormatDefinition('minimal', 'Progress: %percent%%');

$bar = new ProgressBar($output, 3);
$bar->setFormat('minimal');
```

This code defines a new `minimal` format that you can then use for your progress bars:

Listing 25-11
```
1 Progress: 0%
2 Progress: 33%
3 Progress: 100%
```

It is almost always better to redefine built-in formats instead of creating new ones as that allows the display to automatically vary based on the verbosity flag of the command.

When defining a new style that contains placeholders that are only available when the maximum number of steps is known, you should create a `_nomax` variant:

Listing 25-12
```
1 ProgressBar::setFormatDefinition('minimal', '%percent%% %remaining%');
2 ProgressBar::setFormatDefinition('minimal_nomax', '%percent%%');
3
```

```
4  $bar = new ProgressBar($output);
5  $bar->setFormat('minimal');
```

When displaying the progress bar, the format will automatically be set to `minimal_nomax` if the bar does not have a maximum number of steps like in the example above.

A format can contain any valid ANSI codes and can also use the Symfony-specific way to set colors:

Listing 25-13
```
ProgressBar::setFormatDefinition(
    'minimal',
    '<info>%percent%</info>\033[32m%\033[0m <fg=white;bg=blue>%remaining%</>'
);
```

A format can span more than one line; that's very useful when you want to display more contextual information alongside the progress bar (see the example at the beginning of this article).

Bar Settings

Amongst the placeholders, `bar` is a bit special as all the characters used to display it can be customized:

Listing 25-14
```
1  // the finished part of the bar
2  $progress->setBarCharacter('<comment>=</comment>');
3
4  // the unfinished part of the bar
5  $progress->setEmptyBarCharacter(' ');
6
7  // the progress character
8  $progress->setProgressCharacter('|');
9
10 // the bar width
11 $progress->setBarWidth(50);
```

For performance reasons, be careful if you set the total number of steps to a high number. For example, if you're iterating over a large number of items, consider setting the redraw frequency to a higher value by calling *`setRedrawFrequency()`*[9], so it updates on only some iterations:

Listing 25-15
```
1  $progress = new ProgressBar($output, 50000);
2  $progress->start();
3
4  // update every 100 iterations
5  $progress->setRedrawFrequency(100);
6
7  $i = 0;
8  while ($i++ < 50000) {
9      // ... do some work
10
11     $progress->advance();
12 }
```

9. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/ProgressBar.html#method_setRedrawFrequency

Custom Placeholders

If you want to display some information that depends on the progress bar display that are not available in the list of built-in placeholders, you can create your own. Let's see how you can create a `remaining_steps` placeholder that displays the number of remaining steps:

Listing 25-16
```
1 ProgressBar::setPlaceholderFormatterDefinition(
2     'remaining_steps',
3     function (ProgressBar $bar, OutputInterface $output) {
4         return $bar->getMaxSteps() - $bar->getProgress();
5     }
6 );
```

Custom Messages

The `%message%` placeholder allows you to specify a custom message to be displayed with the progress bar. But if you need more than one, just define your own:

Listing 25-17
```
 1 $bar->setMessage('Task starts');
 2 $bar->setMessage('', 'filename');
 3 $bar->start();
 4
 5 $bar->setMessage('Task is in progress...');
 6 while ($file = array_pop($files)) {
 7     $bar->setMessage($filename, 'filename');
 8     $bar->advance();
 9 }
10
11 $bar->setMessage('Task is finished');
12 $bar->setMessage('', 'filename');
13 $bar->finish();
```

For the `filename` to be part of the progress bar, just add the `%filename%` placeholder in your format:

Listing 25-18
```
$bar->setFormat(" %message%\n %current%/%max%\n Working on %filename%");
```

Chapter 26

Progress Helper

The Progress Helper was deprecated in Symfony 2.5 and removed in Symfony 3.0. You should now use the *Progress Bar* instead which is more powerful.

Chapter 27
Question Helper

The *QuestionHelper*[1] provides functions to ask the user for more information. It is included in the default helper set, which you can get by calling *getHelperSet()*[2]:

Listing 27-1
```
$helper = $this->getHelper('question');
```

The Question Helper has a single method *ask()*[3] that needs an *InputInterface*[4] instance as the first argument, an *OutputInterface*[5] instance as the second argument and a *Question*[6] as last argument.

Asking the User for Confirmation

Suppose you want to confirm an action before actually executing it. Add the following to your command:

Listing 27-2
```
1  // ...
2  use Symfony\Component\Console\Input\InputInterface;
3  use Symfony\Component\Console\Output\OutputInterface;
4  use Symfony\Component\Console\Question\ConfirmationQuestion;
5
6  class YourCommand extends Command
7  {
8      // ...
9
10     public function execute(InputInterface $input, OutputInterface $output)
11     {
12         $helper = $this->getHelper('question');
13         $question = new ConfirmationQuestion('Continue with this action?', false);
14
15         if (!$helper->ask($input, $output, $question)) {
```

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/QuestionHelper.html
2. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_getHelperSet
3. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_ask
4. http://api.symfony.com/3.0/Symfony/Component/Console/Input/InputInterface.html
5. http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html
6. http://api.symfony.com/3.0/Symfony/Component/Console/Question/Question.html

```
16                return;
17            }
18        }
19    }
```

In this case, the user will be asked "Continue with this action?". If the user answers with `y` it returns `true` or `false` if they answer with `n`. The second argument to *__construct()*[7] is the default value to return if the user doesn't enter any valid input. If the second argument is not provided, `true` is assumed.

You can customize the regex used to check if the answer means "yes" in the third argument of the constructor. For instance, to allow anything that starts with either `y` or `j`, you would set it to:

Listing 27-3

```
1  $question = new ConfirmationQuestion(
2      'Continue with this action?',
3      false,
4      '/^(y|j)/i'
5  );
```

The regex defaults to `/^y/i`.

Asking the User for Information

You can also ask a question with more than a simple yes/no answer. For instance, if you want to know a bundle name, you can add this to your command:

Listing 27-4

```
1  use Symfony\Component\Console\Question\Question;
2
3  // ...
4  public function execute(InputInterface $input, OutputInterface $output)
5  {
6      // ...
7      $question = new Question('Please enter the name of the bundle', 'AcmeDemoBundle');
8
9      $bundle = $helper->ask($input, $output, $question);
10 }
```

The user will be asked "Please enter the name of the bundle". They can type some name which will be returned by the *ask()*[8] method. If they leave it empty, the default value (`AcmeDemoBundle` here) is returned.

Let the User Choose from a List of Answers

If you have a predefined set of answers the user can choose from, you could use a *ChoiceQuestion*[9] which makes sure that the user can only enter a valid string from a predefined list:

Listing 27-5

```
1  use Symfony\Component\Console\Question\ChoiceQuestion;
2
3  // ...
```

7. http://api.symfony.com/3.0/Symfony/Component/Console/Question/ConfirmationQuestion.html#method___construct
8. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/QuestionHelper.html#method_ask
9. http://api.symfony.com/3.0/Symfony/Component/Console/Question/ChoiceQuestion.html

```
4 public function execute(InputInterface $input, OutputInterface $output)
5 {
6     // ...
7     $helper = $this->getHelper('question');
8     $question = new ChoiceQuestion(
9         'Please select your favorite color (defaults to red)',
10        array('red', 'blue', 'yellow'),
11        0
12    );
13    $question->setErrorMessage('Color %s is invalid.');
14
15    $color = $helper->ask($input, $output, $question);
16    $output->writeln('You have just selected: '.$color);
17
18    // ... do something with the color
19 }
```

The option which should be selected by default is provided with the third argument of the constructor. The default is `null`, which means that no option is the default one.

If the user enters an invalid string, an error message is shown and the user is asked to provide the answer another time, until they enter a valid string or reach the maximum number of attempts. The default value for the maximum number of attempts is `null`, which means infinite number of attempts. You can define your own error message using *`setErrorMessage()`*[10].

Multiple Choices

Sometimes, multiple answers can be given. The `ChoiceQuestion` provides this feature using comma separated values. This is disabled by default, to enable this use *`setMultiselect()`*[11]:

Listing 27-6

```
1 use Symfony\Component\Console\Question\ChoiceQuestion;
2
3 // ...
4 public function execute(InputInterface $input, OutputInterface $output)
5 {
6     // ...
7     $helper = $this->getHelper('question');
8     $question = new ChoiceQuestion(
9         'Please select your favorite colors (defaults to red and blue)',
10        array('red', 'blue', 'yellow'),
11        '0,1'
12    );
13    $question->setMultiselect(true);
14
15    $colors = $helper->ask($input, $output, $question);
16    $output->writeln('You have just selected: ' . implode(', ', $colors));
17 }
```

Now, when the user enters `1,2`, the result will be: `You have just selected: blue, yellow`.

If the user does not enter anything, the result will be: `You have just selected: red, blue`.

10. http://api.symfony.com/3.0/Symfony/Component/Console/Question/ChoiceQuestion.html#method_setErrorMessage
11. http://api.symfony.com/3.0/Symfony/Component/Console/Question/ChoiceQuestion.html#method_setMultiselect

Autocompletion

You can also specify an array of potential answers for a given question. These will be autocompleted as the user types:

Listing 27-7

```
use Symfony\Component\Console\Question\Question;

// ...
public function execute(InputInterface $input, OutputInterface $output)
{
    // ...
    $bundles = array('AcmeDemoBundle', 'AcmeBlogBundle', 'AcmeStoreBundle');
    $question = new Question('Please enter the name of a bundle', 'FooBundle');
    $question->setAutocompleterValues($bundles);

    $name = $helper->ask($input, $output, $question);
}
```

Hiding the User's Response

You can also ask a question and hide the response. This is particularly convenient for passwords:

Listing 27-8

```
use Symfony\Component\Console\Question\Question;

// ...
public function execute(InputInterface $input, OutputInterface $output)
{
    // ...
    $question = new Question('What is the database password?');
    $question->setHidden(true);
    $question->setHiddenFallback(false);

    $password = $helper->ask($input, $output, $question);
}
```

When you ask for a hidden response, Symfony will use either a binary, change stty mode or use another trick to hide the response. If none is available, it will fallback and allow the response to be visible unless you set this behavior to `false` using *`setHiddenFallback()`*[12] like in the example above. In this case, a `RuntimeException` would be thrown.

Validating the Answer

You can even validate the answer. For instance, in a previous example you asked for the bundle name. Following the Symfony naming conventions, it should be suffixed with `Bundle`. You can validate that by using the *`setValidator()`*[13] method:

Listing 27-9

```
use Symfony\Component\Console\Question\Question;

// ...
public function execute(InputInterface $input, OutputInterface $output)
```

12. http://api.symfony.com/3.0/Symfony/Component/Console/Question/Question.html#method_setHiddenFallback
13. http://api.symfony.com/3.0/Symfony/Component/Console/Question/Question.html#method_setValidator

```
5  {
6      // ...
7      $question = new Question('Please enter the name of the bundle', 'AcmeDemoBundle');
8      $question->setValidator(function ($answer) {
9          if ('Bundle' !== substr($answer, -6)) {
10             throw new \RuntimeException(
11                 'The name of the bundle should be suffixed with \'Bundle\''
12             );
13         }
14         return $answer;
15     });
16     $question->setMaxAttempts(2);
17
18     $name = $helper->ask($input, $output, $question);
19 }
```

The `$validator` is a callback which handles the validation. It should throw an exception if there is something wrong. The exception message is displayed in the console, so it is a good practice to put some useful information in it. The callback function should also return the value of the user's input if the validation was successful.

You can set the max number of times to ask with the *`setMaxAttempts()`*[14] method. If you reach this max number it will use the default value. Using `null` means the amount of attempts is infinite. The user will be asked as long as they provide an invalid answer and will only be able to proceed if their input is valid.

Validating a Hidden Response

You can also use a validator with a hidden question:

Listing 27-10

```
1  use Symfony\Component\Console\Question\Question;
2
3  // ...
4  public function execute(InputInterface $input, OutputInterface $output)
5  {
6      // ...
7      $helper = $this->getHelper('question');
8
9      $question = new Question('Please enter your password');
10     $question->setValidator(function ($value) {
11         if (trim($value) == '') {
12             throw new \Exception('The password can not be empty');
13         }
14
15         return $value;
16     });
17     $question->setHidden(true);
18     $question->setMaxAttempts(20);
19
20     $password = $helper->ask($input, $output, $question);
21 }
```

14. http://api.symfony.com/3.0/Symfony/Component/Console/Question/Question.html#method_setMaxAttempts

Testing a Command that Expects Input

If you want to write a unit test for a command which expects some kind of input from the command line, you need to set the helper input stream:

Listing 27-11

```
use Symfony\Component\Console\Helper\QuestionHelper;
use Symfony\Component\Console\Helper\HelperSet;
use Symfony\Component\Console\Tester\CommandTester;

// ...
public function testExecute()
{
    // ...
    $commandTester = new CommandTester($command);

    $helper = $command->getHelper('question');
    $helper->setInputStream($this->getInputStream('Test\\n'));
    // Equals to a user inputting "Test" and hitting ENTER
    // If you need to enter a confirmation, "yes\n" will work

    $commandTester->execute(array('command' => $command->getName()));

    // $this->assertRegExp('/.../', $commandTester->getDisplay());
}

protected function getInputStream($input)
{
    $stream = fopen('php://memory', 'r+', false);
    fputs($stream, $input);
    rewind($stream);

    return $stream;
}
```

By setting the input stream of the `QuestionHelper`, you imitate what the console would do internally with all user input through the CLI. This way you can test any user interaction (even complex ones) by passing an appropriate input stream.

Chapter 28

Table

When building a console application it may be useful to display tabular data:

Listing 28-1

```
+---------------+--------------------------+------------------+
| ISBN          | Title                    | Author           |
+---------------+--------------------------+------------------+
| 99921-58-10-7 | Divine Comedy            | Dante Alighieri  |
| 9971-5-0210-0 | A Tale of Two Cities     | Charles Dickens  |
| 960-425-059-0 | The Lord of the Rings    | J. R. R. Tolkien |
| 80-902734-1-6 | And Then There Were None | Agatha Christie  |
+---------------+--------------------------+------------------+
```

To display a table, use `Table`[1], set the headers, set the rows and then render the table:

Listing 28-2

```
use Symfony\Component\Console\Helper\Table;
// ...

class SomeCommand extends Command
{
    public function execute(InputInterface $input, OutputInterface $output)
    {
        $table = new Table($output);
        $table
            ->setHeaders(array('ISBN', 'Title', 'Author'))
            ->setRows(array(
                array('99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'),
                array('9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'),
                array('960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'),
                array('80-902734-1-6', 'And Then There Were None', 'Agatha Christie'),
            ))
        ;
        $table->render();
    }
}
```

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/Table.html

You can add a table separator anywhere in the output by passing an instance of *TableSeparator*[2] as a row:

Listing 28-3
```
use Symfony\Component\Console\Helper\TableSeparator;

$table->setRows(array(
    array('99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'),
    array('9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'),
    new TableSeparator(),
    array('960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'),
    array('80-902734-1-6', 'And Then There Were None', 'Agatha Christie'),
));
```

Listing 28-4
```
+---------------+--------------------------+------------------+
| ISBN          | Title                    | Author           |
+---------------+--------------------------+------------------+
| 99921-58-10-7 | Divine Comedy            | Dante Alighieri  |
| 9971-5-0210-0 | A Tale of Two Cities     | Charles Dickens  |
+---------------+--------------------------+------------------+
| 960-425-059-0 | The Lord of the Rings    | J. R. R. Tolkien |
| 80-902734-1-6 | And Then There Were None | Agatha Christie  |
+---------------+--------------------------+------------------+
```

The table style can be changed to any built-in styles via *setStyle()*[3]:

Listing 28-5
```
// same as calling nothing
$table->setStyle('default');

// changes the default style to compact
$table->setStyle('compact');
$table->render();
```

This code results in:

Listing 28-6
```
ISBN          Title                    Author
99921-58-10-7 Divine Comedy            Dante Alighieri
9971-5-0210-0 A Tale of Two Cities     Charles Dickens
960-425-059-0 The Lord of the Rings    J. R. R. Tolkien
80-902734-1-6 And Then There Were None Agatha Christie
```

You can also set the style to `borderless`:

Listing 28-7
```
$table->setStyle('borderless');
$table->render();
```

which outputs:

Listing 28-8
```
=============== ========================== ==================
 ISBN            Title                      Author
=============== ========================== ==================
 99921-58-10-7   Divine Comedy              Dante Alighieri
 9971-5-0210-0   A Tale of Two Cities       Charles Dickens
 960-425-059-0   The Lord of the Rings      J. R. R. Tolkien
```

2. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableSeparator.html
3. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/Table.html#method_setStyle

```
7   80-902734-1-6   And Then There Were None   Agatha Christie
8  =============== ========================== ==================
```

If the built-in styles do not fit your need, define your own:

Listing 28-9
```
use Symfony\Component\Console\Helper\TableStyle;

// by default, this is based on the default style
$style = new TableStyle();

// customize the style
$style
    ->setHorizontalBorderChar('<fg=magenta>|</>')
    ->setVerticalBorderChar('<fg=magenta>-</>')
    ->setCrossingChar(' ')
;

// use the style for this table
$table->setStyle($style);
```

Here is a full list of things you can customize:

- *setPaddingChar()*[4]
- *setHorizontalBorderChar()*[5]
- *setVerticalBorderChar()*[6]
- *setCrossingChar()*[7]
- *setCellHeaderFormat()*[8]
- *setCellRowFormat()*[9]
- *setBorderFormat()*[10]
- *setPadType()*[11]

You can also register a style globally:

Listing 28-10
```
// register the style under the colorful name
Table::setStyleDefinition('colorful', $style);

// use it for a table
$table->setStyle('colorful');
```

This method can also be used to override a built-in style.

Spanning Multiple Columns and Rows

To make a table cell that spans multiple columns you can use a *TableCell*[12]:

4. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setPaddingChar
5. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setHorizontalBorderChar
6. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setVerticalBorderChar
7. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setCrossingChar
8. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setCellHeaderFormat
9. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setCellRowFormat
10. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setBorderFormat
11. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableStyle.html#method_setPadType
12. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/TableCell.html

Listing 28-11

```
use Symfony\Component\Console\Helper\Table;
use Symfony\Component\Console\Helper\TableSeparator;
use Symfony\Component\Console\Helper\TableCell;

$table = new Table($output);
$table
    ->setHeaders(array('ISBN', 'Title', 'Author'))
    ->setRows(array(
        array('99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'),
        new TableSeparator(),
        array(new TableCell('This value spans 3 columns.', array('colspan' => 3))),
    ))
;
$table->render();
```

This results in:

Listing 28-12

```
+---------------+---------------+-----------------+
| ISBN          | Title         | Author          |
+---------------+---------------+-----------------+
| 99921-58-10-7 | Divine Comedy | Dante Alighieri |
+---------------+---------------+-----------------+
| This value spans 3 columns.                     |
+---------------+---------------+-----------------+
```

You can create a multiple-line page title using a header cell that spans the enire table width:

Listing 28-13

```
$table->setHeaders(array(
    array(new TableCell('Main table title', array('colspan' => 3))),
    array('ISBN', 'Title', 'Author'),
))
// ...
```

This generates:

Listing 28-14

```
+-------+-------+--------+
| Main table title       |
+-------+-------+--------+
| ISBN  | Title | Author |
+-------+-------+--------+
| ...                    |
+-------+-------+--------+
```

In a similar way you can span multiple rows:

Listing 28-15

```
use Symfony\Component\Console\Helper\Table;
use Symfony\Component\Console\Helper\TableCell;

$table = new Table($output);
$table
    ->setHeaders(array('ISBN', 'Title', 'Author'))
    ->setRows(array(
        array(
            '978-0521567817',
```

```
10                 'De Monarchia',
11                 new TableCell("Dante Alighieri\nspans multiple rows", array('rowspan' => 2)),
12             ),
13             array('978-0804169127', 'Divine Comedy'),
14         ))
15 ;
16 $table->render();
```

This outputs:

Listing 28-16

```
1 +---------------+---------------+---------------------+
2 | ISBN          | Title         | Author              |
3 +---------------+---------------+---------------------+
4 | 978-0521567817 | De Monarchia  | Dante Alighieri     |
5 | 978-0804169127 | Divine Comedy | spans multiple rows |
6 +---------------+---------------+---------------------+
```

You can use the `colspan` and `rowspan` options at the same time which allows you to create any table layout you may wish.

Chapter 29
Table Helper

The Table Helper was deprecated in Symfony 2.5 and removed in Symfony 3.0. You should now use the *Table* class instead which is more powerful.

Chapter 30

Debug Formatter Helper

The *DebugFormatterHelper*[1] provides functions to output debug information when running an external program, for instance a process or HTTP request. For example, if you used it to output the results of running `ls -la` on a UNIX system, it might output something like this:

```
RUN  Running ls

OUT  total 0
OUT  drwxr-xr-x   5 weaverryan  staff  170 Dec 16 08:36 .
OUT  drwxr-xr-x  16 weaverryan  staff  544 Dec 16 07:38 ..
OUT  drwxr-xr-x   2 weaverryan  staff   68 Dec 16 08:36 Symfony3
OUT  -rw-r--r--   1 weaverryan  staff    0 Dec 16 07:38 big_data.csv
OUT  -rw-r--r--   1 weaverryan  staff    0 Dec 16 07:38 the_cloud.zip
OUT

RES  Finishing the command
```

Using the debug_formatter

The formatter is included in the default helper set and you can get it by calling *getHelper()*[2]:

Listing 30-1
```
$debugFormatter = $this->getHelper('debug_formatter');
```

The formatter accepts strings and returns a formatted string, which you then output to the console (or even log the information or do anything else).

All methods of this helper have an identifier as the first argument. This is a unique value for each program. This way, the helper can debug information for multiple programs at the same time. When using the *Process component*, you probably want to use *spl_object_hash*[3].

1. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/DebugFormatterHelper.html
2. http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html#method_getHelper
3. http://php.net/manual/en/function.spl-object-hash.php

This information is often too verbose to be shown by default. You can use verbosity levels to only show it when in debugging mode (`-vvv`).

Starting a Program

As soon as you start a program, you can use *start()*[4] to display information that the program is started:

Listing 30-2
```
// ...
$process = new Process(...);

$output->writeln($debugFormatter->start(
    spl_object_hash($process),
    'Some process description'
));

$process->run();
```

This will output:

Listing 30-3
```
RUN Some process description
```

You can tweak the prefix using the third argument:

Listing 30-4
```
$output->writeln($debugFormatter->start(
    spl_object_hash($process),
    'Some process description',
    'STARTED'
));
// will output:
//  STARTED Some process description
```

Output Progress Information

Some programs give output while they are running. This information can be shown using *progress()*[5]:

Listing 30-5
```
use Symfony\Component\Process\Process;

// ...
$process = new Process(...);

$process->run(function ($type, $buffer) use ($output, $debugFormatter, $process) {
    $output->writeln(
        $debugFormatter->progress(
            spl_object_hash($process),
            $buffer,
            Process::ERR === $type
        )
```

4. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/DebugFormatterHelper.html#method_start
5. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/DebugFormatterHelper.html#method_progress

```
13     );
14 });
15 // ...
```

In case of success, this will output:

Listing 30-6
```
1 OUT The output of the process
```

And this in case of failure:

Listing 30-7
```
1 ERR The output of the process
```

The third argument is a boolean which tells the function if the output is error output or not. When `true`, the output is considered error output.

The fourth and fifth argument allow you to override the prefix for the normal output and error output respectively.

Stopping a Program

When a program is stopped, you can use *stop()*[6] to notify this to the users:

Listing 30-8
```
1 // ...
2 $output->writeln(
3     $debugFormatter->stop(
4         spl_object_hash($process),
5         'Some command description',
6         $process->isSuccessfull()
7     )
8 );
```

This will output:

Listing 30-9
```
1 RES Some command description
```

In case of failure, this will be in red and in case of success it will be green.

Using multiple Programs

As said before, you can also use the helper to display more programs at the same time. Information about different programs will be shown in different colors, to make it clear which output belongs to which command.

6. http://api.symfony.com/3.0/Symfony/Component/Console/Helper/DebugFormatterHelper.html#method_stop

Chapter 31

The CssSelector Component

> The CssSelector component converts CSS selectors to XPath expressions.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/css-selector` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/css-selector*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

Why to Use CSS selectors?

When you're parsing an HTML or an XML document, by far the most powerful method is XPath.

XPath expressions are incredibly flexible, so there is almost always an XPath expression that will find the element you need. Unfortunately, they can also become very complicated, and the learning curve is steep. Even common operations (such as finding an element with a particular class) can require long and unwieldy expressions.

Many developers -- particularly web developers -- are more comfortable using CSS selectors to find elements. As well as working in stylesheets, CSS selectors are used in JavaScript with the `querySelectorAll` function and in popular JavaScript libraries such as jQuery, Prototype and MooTools.

1. `https://packagist.org/packages/symfony/css-selector`

CSS selectors are less powerful than XPath, but far easier to write, read and understand. Since they are less powerful, almost all CSS selectors can be converted to an XPath equivalent. This XPath expression can then be used with other functions and classes that use XPath to find elements in a document.

The CssSelector Component

The component's only goal is to convert CSS selectors to their XPath equivalents, using *toXPath()*[2]:

Listing 31-1
```
use Symfony\Component\CssSelector\CssSelectorConverter;

$converter = new CssSelectorConverter();
var_dump($converter->toXPath('div.item > h4 > a'));
```

This gives the following output:

Listing 31-2
```
1 descendant-or-self::div[@class and contains(concat(' ',normalize-space(@class), ' '), '
  item ')]/h4/a
```

You can use this expression with, for instance, *DOMXPath*[3] or *SimpleXMLElement*[4] to find elements in a document.

The *Crawler::filter()*[5] method uses the CssSelector component to find elements based on a CSS selector string. See the *The DomCrawler Component* for more details.

Limitations of the CssSelector Component

Not all CSS selectors can be converted to XPath equivalents.

There are several CSS selectors that only make sense in the context of a web-browser.

- link-state selectors: `:link`, `:visited`, `:target`
- selectors based on user action: `:hover`, `:focus`, `:active`
- UI-state selectors: `:invalid`, `:indeterminate` (however, `:enabled`, `:disabled`, `:checked` and `:unchecked` are available)

Pseudo-elements (`:before`, `:after`, `:first-line`, `:first-letter`) are not supported because they select portions of text rather than elements.

Several pseudo-classes are not yet supported:

- `*:first-of-type`, `*:last-of-type`, `*:nth-of-type`, `*:nth-last-of-type`, `*:only-of-type`. (These work with an element name (e.g. `li:first-of-type`) but not with `*`.

2. http://api.symfony.com/3.0/Symfony/Component/CssSelector/CssSelectorConverter.html#method_toXPath
3. http://php.net/manual/en/class.domxpath.php
4. http://php.net/manual/en/class.simplexmlelement.php
5. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_filter

Chapter 32
The Debug Component

> The Debug component provides tools to ease debugging PHP code.

Installation

You can install the component in many different ways:

- *Install it via Composer* (`symfony/debug` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/debug*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The Debug component provides several tools to help you debug PHP code. Enabling them all is as easy as it can get:

Listing 32-1
```
use Symfony\Component\Debug\Debug;

Debug::enable();
```

The *`enable()`*[2] method registers an error handler, an exception handler and *a special class loader*.

Read the following sections for more information about the different available tools.

1. `https://packagist.org/packages/symfony/debug`
2. `http://api.symfony.com/3.0/Symfony/Component/Debug/Debug.html#method_enable`

You should never enable the debug tools in a production environment as they might disclose sensitive information to the user.

Enabling the Error Handler

The *ErrorHandler*[3] class catches PHP errors and converts them to exceptions (of class *ErrorException*[4] or *FatalErrorException*[5] for PHP fatal errors):

Listing 32-2
```
use Symfony\Component\Debug\ErrorHandler;

ErrorHandler::register();
```

Enabling the Exception Handler

The *ExceptionHandler*[6] class catches uncaught PHP exceptions and converts them to a nice PHP response. It is useful in debug mode to replace the default PHP/XDebug output with something prettier and more useful:

Listing 32-3
```
use Symfony\Component\Debug\ExceptionHandler;

ExceptionHandler::register();
```

If the *HttpFoundation component* is available, the handler uses a Symfony Response object; if not, it falls back to a regular PHP response.

3. http://api.symfony.com/3.0/Symfony/Component/Debug/ErrorHandler.html
4. http://php.net/manual/en/class.errorexception.php
5. http://api.symfony.com/3.0/Symfony/Component/Debug/Exception/FatalErrorException.html
6. http://api.symfony.com/3.0/Symfony/Component/Debug/ExceptionHandler.html

Chapter 33
Debugging a Class Loader

The *DebugClassLoader*[1] attempts to throw more helpful exceptions when a class isn't found by the registered autoloaders. All autoloaders that implement a `findFile()` method are replaced with a `DebugClassLoader` wrapper.

Using the `DebugClassLoader` is as easy as calling its static *enable()*[2] method:

Listing 33-1
```
use Symfony\Component\Debug\DebugClassLoader;

DebugClassLoader::enable();
```

1. http://api.symfony.com/3.0/Symfony/Component/Debug/DebugClassLoader.html
2. http://api.symfony.com/3.0/Symfony/Component/Debug/DebugClassLoader.html#method_enable

Chapter 34
The DependencyInjection Component

The DependencyInjection component allows you to standardize and centralize the way objects are constructed in your application.

For an introduction to Dependency Injection and service containers see *Service Container*.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/dependency-injection` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/dependency-injection*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Basic Usage

You might have a simple class like the following `Mailer` that you want to make available as a service:

Listing 34-1

```
class Mailer
{
    private $transport;

    public function __construct()
    {
        $this->transport = 'sendmail';
    }

```

1. `https://packagist.org/packages/symfony/dependency-injection`

```
10      // ...
11  }
```

You can register this in the container as a service:

Listing 34-2
```
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->register('mailer', 'Mailer');
```

An improvement to the class to make it more flexible would be to allow the container to set the `transport` used. If you change the class so this is passed into the constructor:

Listing 34-3
```
1  class Mailer
2  {
3      private $transport;
4
5      public function __construct($transport)
6      {
7          $this->transport = $transport;
8      }
9
10     // ...
11 }
```

Then you can set the choice of transport in the container:

Listing 34-4
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2
3  $container = new ContainerBuilder();
4  $container
5      ->register('mailer', 'Mailer')
6      ->addArgument('sendmail');
```

This class is now much more flexible as you have separated the choice of transport out of the implementation and into the container.

Which mail transport you have chosen may be something other services need to know about. You can avoid having to change it in multiple places by making it a parameter in the container and then referring to this parameter for the `Mailer` service's constructor argument:

Listing 34-5
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2
3  $container = new ContainerBuilder();
4  $container->setParameter('mailer.transport', 'sendmail');
5  $container
6      ->register('mailer', 'Mailer')
7      ->addArgument('%mailer.transport%');
```

Now that the `mailer` service is in the container you can inject it as a dependency of other classes. If you have a `NewsletterManager` class like this:

Listing 34-6
```
1  class NewsletterManager
2  {
3      private $mailer;
4
```

```
5       public function __construct(\Mailer $mailer)
6       {
7           $this->mailer = $mailer;
8       }
9
10      // ...
11  }
```

When defining the `newsletter_manager` service, the `mailer` service does not exist yet. Use the `Reference` class to tell the container to inject the `mailer` service when it initializes the newsletter manager:

Listing 34-7

```
1   use Symfony\Component\DependencyInjection\ContainerBuilder;
2   use Symfony\Component\DependencyInjection\Reference;
3
4   $container = new ContainerBuilder();
5
6   $container->setParameter('mailer.transport', 'sendmail');
7   $container
8       ->register('mailer', 'Mailer')
9       ->addArgument('%mailer.transport%');
10
11  $container
12      ->register('newsletter_manager', 'NewsletterManager')
13      ->addArgument(new Reference('mailer'));
```

If the `NewsletterManager` did not require the `Mailer` and injecting it was only optional then you could use setter injection instead:

Listing 34-8

```
1   class NewsletterManager
2   {
3       private $mailer;
4
5       public function setMailer(\Mailer $mailer)
6       {
7           $this->mailer = $mailer;
8       }
9
10      // ...
11  }
```

You can now choose not to inject a `Mailer` into the `NewsletterManager`. If you do want to though then the container can call the setter method:

Listing 34-9

```
1   use Symfony\Component\DependencyInjection\ContainerBuilder;
2   use Symfony\Component\DependencyInjection\Reference;
3
4   $container = new ContainerBuilder();
5
6   $container->setParameter('mailer.transport', 'sendmail');
7   $container
8       ->register('mailer', 'Mailer')
9       ->addArgument('%mailer.transport%');
10
11  $container
```

```
12     ->register('newsletter_manager', 'NewsletterManager')
13     ->addMethodCall('setMailer', array(new Reference('mailer')));
```

You could then get your `newsletter_manager` service from the container like this:

Listing 34-10
```
1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2
3 $container = new ContainerBuilder();
4
5 // ...
6
7 $newsletterManager = $container->get('newsletter_manager');
```

Avoiding your Code Becoming Dependent on the Container

Whilst you can retrieve services from the container directly it is best to minimize this. For example, in the `NewsletterManager` you injected the `mailer` service in rather than asking for it from the container. You could have injected the container in and retrieved the `mailer` service from it but it would then be tied to this particular container making it difficult to reuse the class elsewhere.

You will need to get a service from the container at some point but this should be as few times as possible at the entry point to your application.

Setting up the Container with Configuration Files

As well as setting up the services using PHP as above you can also use configuration files. This allows you to use XML or YAML to write the definitions for the services rather than using PHP to define the services as in the above examples. In anything but the smallest applications it makes sense to organize the service definitions by moving them into one or more configuration files. To do this you also need to install *the Config component*.

Loading an XML config file:

Listing 34-11
```
1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2 use Symfony\Component\Config\FileLocator;
3 use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
4
5 $container = new ContainerBuilder();
6 $loader = new XmlFileLoader($container, new FileLocator(__DIR__));
7 $loader->load('services.xml');
```

Loading a YAML config file:

Listing 34-12
```
1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2 use Symfony\Component\Config\FileLocator;
3 use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
4
5 $container = new ContainerBuilder();
6 $loader = new YamlFileLoader($container, new FileLocator(__DIR__));
7 $loader->load('services.yml');
```

If you want to load YAML config files then you will also need to install *the Yaml component*.

If you *do* want to use PHP to create the services then you can move this into a separate config file and load it in a similar way:

Listing 34-13
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\PhpFileLoader;

$container = new ContainerBuilder();
$loader = new PhpFileLoader($container, new FileLocator(__DIR__));
$loader->load('services.php');
```

You can now set up the `newsletter_manager` and `mailer` services using config files:

Listing 34-14
```yaml
parameters:
    # ...
    mailer.transport: sendmail

services:
    mailer:
        class:     Mailer
        arguments: ['%mailer.transport%']
    newsletter_manager:
        class:     NewsletterManager
        calls:
            - [setMailer, ['@mailer']]
```

Chapter 35
Types of Injection

Making a class's dependencies explicit and requiring that they be injected into it is a good way of making a class more reusable, testable and decoupled from others.

There are several ways that the dependencies can be injected. Each injection point has advantages and disadvantages to consider, as well as different ways of working with them when using the service container.

Constructor Injection

The most common way to inject dependencies is via a class's constructor. To do this you need to add an argument to the constructor signature to accept the dependency:

Listing 35-1

```
class NewsletterManager
{
    protected $mailer;

    public function __construct(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

You can specify what service you would like to inject into this in the service container configuration:

Listing 35-2

```
services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     NewsletterManager
        arguments: ['@my_mailer']
```

Type hinting the injected object means that you can be sure that a suitable dependency has been injected. By type-hinting, you'll get a clear error immediately if an unsuitable dependency is injected. By type hinting using an interface rather than a class you can make the choice of dependency more flexible. And assuming you only use methods defined in the interface, you can gain that flexibility and still safely use the object.

There are several advantages to using constructor injection:

- If the dependency is a requirement and the class cannot work without it then injecting it via the constructor ensures it is present when the class is used as the class cannot be constructed without it.
- The constructor is only ever called once when the object is created, so you can be sure that the dependency will not change during the object's lifetime.

These advantages do mean that constructor injection is not suitable for working with optional dependencies. It is also more difficult to use in combination with class hierarchies: if a class uses constructor injection then extending it and overriding the constructor becomes problematic.

Setter Injection

Another possible injection point into a class is by adding a setter method that accepts the dependency:

Listing 35-3

```
class NewsletterManager
{
    protected $mailer;

    public function setMailer(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Listing 35-4

```
services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     NewsletterManager
        calls:
            - [setMailer, ['@my_mailer']]
```

This time the advantages are:

- Setter injection works well with optional dependencies. If you do not need the dependency, then just do not call the setter.
- You can call the setter multiple times. This is particularly useful if the method adds the dependency to a collection. You can then have a variable number of dependencies.

The disadvantages of setter injection are:

- The setter can be called more than just at the time of construction so you cannot be sure the dependency is not replaced during the lifetime of the object (except by explicitly writing the setter method to check if it has already been called).

- You cannot be sure the setter will be called and so you need to add checks that any required dependencies are injected.

Property Injection

Another possibility is just setting public fields of the class directly:

Listing 35-5
```
1 class NewsletterManager
2 {
3     public $mailer;
4
5     // ...
6 }
```

Listing 35-6
```
1 services:
2      my_mailer:
3          # ...
4      newsletter_manager:
5          class: NewsletterManager
6          properties:
7              mailer: '@my_mailer'
```

There are mainly only disadvantages to using property injection, it is similar to setter injection but with these additional important problems:

- You cannot control when the dependency is set at all, it can be changed at any point in the object's lifetime.
- You cannot use type hinting so you cannot be sure what dependency is injected except by writing into the class code to explicitly test the class instance before using it.

But, it is useful to know that this can be done with the service container, especially if you are working with code that is out of your control, such as in a third party library, which uses public properties for its dependencies.

Chapter 36

Introduction to Parameters

You can define parameters in the service container which can then be used directly or as part of service definitions. This can help to separate out values that you will want to change more regularly.

Getting and Setting Container Parameters

Working with container parameters is straightforward using the container's accessor methods for parameters. You can check if a parameter has been defined in the container with:

Listing 36-1
```
$container->hasParameter('mailer.transport');
```

You can retrieve a parameter set in the container with:

Listing 36-2
```
$container->getParameter('mailer.transport');
```

and set a parameter in the container with:

Listing 36-3
```
$container->setParameter('mailer.transport', 'sendmail');
```

The used `.` notation is just a Symfony convention to make parameters easier to read. Parameters are just flat key-value elements, they can't be organized into a nested array

You can only set a parameter before the container is compiled. To learn more about compiling the container see *Compiling the Container*.

Parameters in Configuration Files

You can also use the `parameters` section of a config file to set parameters:

Listing 36-4

```
1 parameters:
2     mailer.transport: sendmail
```

As well as retrieving the parameter values directly from the container you can use them in the config files. You can refer to parameters elsewhere by surrounding them with percent (%) signs, e.g. `%mailer.transport%`. One use for this is to inject the values into your services. This allows you to configure different versions of services between applications or multiple services based on the same class but configured differently within a single application. You could inject the choice of mail transport into the `Mailer` class directly. But declaring it as a parameter makes it easier to change rather than being tied up and hidden with the service definition:

Listing 36-5

```
1 parameters:
2     mailer.transport: sendmail
3
4 services:
5     mailer:
6         class:     Mailer
7         arguments: ['%mailer.transport%']
```

The values between `parameter` tags in XML configuration files are not trimmed.

This means that the following configuration sample will have the value `\n sendmail\n`:

Listing 36-6

```
1 <parameter key="mailer.transport">
2     sendmail
3 </parameter>
```

In some cases (for constants or class names), this could throw errors. In order to prevent this, you must always inline your parameters as follow:

Listing 36-7

```
1 <parameter key="mailer.transport">sendmail</parameter>
```

If you were using this elsewhere as well, then you would only need to change the parameter value in one place if needed.

The percent sign inside a parameter or argument, as part of the string, must be escaped with another percent sign:

Listing 36-8

```
1 arguments: ['http://symfony.com/?foo=%%s&bar=%%d']
```

Array Parameters

Parameters do not need to be flat strings, they can also contain array values. For the XML format, you need to use the `type="collection"` attribute for all parameters that are arrays.

Listing 36-9

```
1 parameters:
2     my_mailer.gateways:
3         - mail1
4         - mail2
```

```
5           - mail3
6       my_multilang.language_fallback:
7           en:
8               - en
9               - fr
10          fr:
11              - fr
12              - en
```

Constants as Parameters

The container also has support for setting PHP constants as parameters. To take advantage of this feature, map the name of your constant to a parameter key and define the type as `constant`.

Listing 36-10

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <container xmlns="http://symfony.com/schema/dic/services"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/
5  dic/services/services-1.0.xsd">
6
7      <parameters>
8          <parameter key="global.constant.value" type="constant">GLOBAL_CONSTANT</parameter>
9          <parameter key="my_class.constant.value"
10 type="constant">My_Class::CONSTANT_NAME</parameter>
       </parameters>
   </container>
```

This does not work for YAML configurations. If you're using YAML, you can import an XML file to take advantage of this functionality:

Listing 36-11

```
1  imports:
2      - { resource: parameters.xml }
```

PHP Keywords in XML

By default, `true`, `false` and `null` in XML are converted to the PHP keywords (respectively `true`, `false` and `null`):

Listing 36-12

```
1  <parameters>
2      <parameter key="mailer.send_all_in_once">false</parameter>
3  </parameters>
4
5  <!-- after parsing
6  $container->getParameter('mailer.send_all_in_once'); // returns false
7  -->
```

To disable this behavior, use the `string` type:

Listing 36-13

```xml
<parameters>
    <parameter key="mailer.some_parameter" type="string">true</parameter>
</parameters>

<!-- after parsing
$container->getParameter('mailer.some_parameter'); // returns "true"
-->
```

This is not available for YAML and PHP, because they already have built-in support for the PHP keywords.

Chapter 37

Working with Container Service Definitions

Service definitions are the instructions describing how the container should build a service. They are not the actual services used by your applications

Getting and Setting Service Definitions

There are some helpful methods for working with the service definitions.

To find out if there is a definition for a service id:

Listing 37-1
```
$container->hasDefinition($serviceId);
```

This is useful if you only want to do something if a particular definition exists.

You can retrieve a definition with:

Listing 37-2
```
$container->getDefinition($serviceId);
```

or:

Listing 37-3
```
$container->findDefinition($serviceId);
```

which unlike `getDefinition()` also resolves aliases so if the `$serviceId` argument is an alias you will get the underlying definition.

The service definitions themselves are objects so if you retrieve a definition with these methods and make changes to it these will be reflected in the container. If, however, you are creating a new definition then you can add it to the container using:

Listing 37-4
```
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Acme\Service\MyService');
$container->setDefinition('acme.my_service', $definition);
```

Registering service definitions is so common that the container provides a shortcut method called `register()`:

Listing 37-5
```
$container->register('acme.my_service', 'Acme\Service\MyService');
```

Working with a Definition

Creating a New Definition

In addition to manipulating and retrieving existing definitions, you can also define new service definitions with the *`Definition`*[1] class.

Class

The first optional argument of the `Definition` class is the fully qualified class name of the object returned when the service is fetched from the container:

Listing 37-6
```
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Acme\Service\MyService');
```

If the class is unknown when instantiating the `Definition` class, use the `setClass()` method to set it later:

Listing 37-7
```
$definition->setClass('Acme\Service\MyService');
```

To find out what class is set for a definition:

Listing 37-8
```
$class = $definition->getClass();
// $class = 'Acme\Service\MyService'
```

Constructor Arguments

The second optional argument of the `Definition` class is an array with the arguments passed to the constructor of the object returned when the service is fetched from the container:

Listing 37-9
```
1 use Symfony\Component\DependencyInjection\Definition;
2 
3 $definition = new Definition(
4     'Acme\Service\MyService',
5     array('argument1' => 'value1', 'argument2' => 'value2')
6 );
```

If the arguments are unknown when instantiating the `Definition` class or if you want to add new arguments, use the `addArgument()` method, which adds them at the end of the arguments array:

Listing 37-10
```
$definition->addArgument($argument);
```

To get an array of the constructor arguments for a definition you can use:

Listing 37-11
```
$definition->getArguments();
```

1. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Definition.html

or to get a single argument by its position:

Listing 37-12
```
$definition->getArgument($index);
// e.g. $definition->getArgument(0) for the first argument
```

The argument can be a string, an array or a service parameter by using the `%parameter_name%` syntax:

Listing 37-13
```
$definition->addArgument('%kernel_debug%');
```

If the argument is another service, don't use the `get()` method to fetch it, because it won't be available when defining services. Instead, use the *`Reference`*[2] class to get a reference to the service which will be available once the service container is fully built:

Listing 37-14
```
1 use Symfony\Component\DependencyInjection\Reference;
2
3 // ...
4
5 $definition->addArgument(new Reference('service_id'));
```

In a similar way you can replace an already set argument by index using:

Listing 37-15
```
$definition->replaceArgument($index, $argument);
```

You can also replace all the arguments (or set some if there are none) with an array of arguments:

Listing 37-16
```
$definition->setArguments($arguments);
```

Method Calls

If the service you are working with uses setter injection then you can manipulate any method calls in the definitions as well.

You can get an array of all the method calls with:

Listing 37-17
```
$definition->getMethodCalls();
```

Add a method call with:

Listing 37-18
```
$definition->addMethodCall($method, $arguments);
```

Where `$method` is the method name and `$arguments` is an array of the arguments to call the method with. The arguments can be strings, arrays, parameters or service ids as with the constructor arguments.

You can also replace any existing method calls with an array of new ones with:

Listing 37-19
```
$definition->setMethodCalls($methodCalls);
```

There are more examples of specific ways of working with definitions in the PHP code blocks of the configuration examples on pages such as *Using a Factory to Create Services* and *Managing Common Dependencies with Parent Services*.

The methods here that change service definitions can only be used before the container is compiled. Once the container is compiled you cannot manipulate service definitions further. To learn more about compiling the container see *Compiling the Container*.

2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Reference.html

Requiring Files

There might be use cases when you need to include another file just before the service itself gets loaded. To do so, you can use the `setFile()`[3] method:

Listing 37-20
```
$definition->setFile('/src/path/to/file/foo.php');
```

Notice that Symfony will internally call the PHP statement `require_once`, which means that your file will be included only once per request.

3. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Definition.html#method_setFile

Chapter 38

Defining Services Dependencies Automatically (Autowiring)

New in version 2.8: Support for autowiring services was introduced in Symfony 2.8.

Autowiring allows to register services in the container with minimal configuration. It automatically resolves the service dependencies based on the constructor's typehint which is useful in the field of *Rapid Application Development*[1], when designing prototypes in early stages of large projects. It makes it easy to register a service graph and eases refactoring.

Imagine you're building an API to publish statuses on a Twitter feed, obfuscated with *ROT13*[2] (a special case of the Caesar cipher).

Start by creating a ROT13 transformer class:

Listing 38-1

```
// src/AppBundle/Rot13Transformer.php
namespace AppBundle;

class Rot13Transformer
{
    public function transform($value)
    {
        return str_rot13($value);
    }
}
```

And now a Twitter client using this transformer:

Listing 38-2

```
// src/AppBundle/TwitterClient.php
namespace AppBundle;

class TwitterClient
{
```

1. https://en.wikipedia.org/wiki/Rapid_application_development
2. https://en.wikipedia.org/wiki/ROT13

```
6       private $transformer;
7
8       public function __construct(Rot13Transformer $transformer)
9       {
10          $this->transformer = $transformer;
11      }
12
13      public function tweet($user, $key, $status)
14      {
15          $transformedStatus = $this->transformer->transform($status);
16
17          // ... connect to Twitter and send the encoded status
18      }
19  }
```

The DependencyInjection component will be able to automatically register the dependencies of this `TwitterClient` class when the `twitter_client` service is marked as autowired:

Listing 38-3

```
1  # app/config/services.yml
2  services:
3      twitter_client:
4          class:    'AppBundle\TwitterClient'
5          autowire: true
```

The autowiring subsystem will detect the dependencies of the `TwitterClient` class by parsing its constructor. For instance it will find here an instance of a `Rot13Transformer` as dependency. If an existing service definition (and only one – see below) is of the required type, this service will be injected. If it's not the case (like in this example), the subsystem is smart enough to automatically register a private service for the `Rot13Transformer` class and set it as first argument of the `twitter_client` service. Again, it can work only if there is one class of the given type. If there are several classes of the same type, you must use an explicit service definition or register a default implementation.

As you can see, the autowiring feature drastically reduces the amount of configuration required to define a service. No more arguments section! It also makes it easy to change the dependencies of the `TwitterClient` class: just add or remove typehinted arguments in the constructor and you are done. There is no need anymore to search and edit related service definitions.

Here is a typical controller using the `twitter_client` service:

Listing 38-4

```
1   // src/AppBundle/Controller/DefaultController.php
2   namespace AppBundle\Controller;
3
4   use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
5   use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
6   use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7   use Symfony\Component\HttpFoundation\Request;
8   use Symfony\Component\HttpFoundation\Response;
9   use Symfony\Component\HttpKernel\Exception\BadRequestHttpException;
10
11  class DefaultController extends Controller
12  {
13      /**
14       * @Route("/tweet")
15       * @Method("POST")
16       */
17      public function tweetAction(Request $request)
18      {
```

```
19          $user = $request->request->get('user');
20          $key = $request->request->get('key');
21          $status = $request->request->get('status');
22
23          if (!$user || !$key || !$status) {
24              throw new BadRequestHttpException();
25          }
26
27          $this->get('twitter_client')->tweet($user, $key, $status);
28
29          return new Response('OK');
30      }
31  }
```

You can give the API a try using `curl`:

Listing 38-5

```
1  $ curl -d "user=kevin&key=ABCD&status=Hello" http://localhost:8000/tweet
```

It should return `OK`.

Working with Interfaces

You might also find yourself using abstractions instead of implementations (especially in grown applications) as it allows to easily replace some dependencies without modifying the class depending of them.

To follow this best practice, constructor arguments must be typehinted with interfaces and not concrete classes. It allows to replace easily the current implementation if necessary. It also allows to use other transformers.

Let's introduce a `TransformerInterface`:

Listing 38-6

```
1  // src/AppBundle/TransformerInterface.php
2  namespace AppBundle;
3
4  interface TransformerInterface
5  {
6      public function transform($value);
7  }
```

Then edit `Rot13Transformer` to make it implementing the new interface:

Listing 38-7

```
1  // ...
2
3  class Rot13Transformer implements TransformerInterface
4
5  // ...
```

And update `TwitterClient` to depend of this new interface:

Listing 38-8

```
1  class TwitterClient
2  {
3      // ...
4
```

```
5      public function __construct(TransformerInterface $transformer)
6      {
7          // ...
8      }
9
10     // ...
11 }
```

Finally the service definition must be updated because, obviously, the autowiring subsystem isn't able to find itself the interface implementation to register:

Listing 38-9
```
1  # app/config/services.yml
2  services:
3      rot13_transformer:
4          class: 'AppBundle\Rot13Transformer'
5
6      twitter_client:
7          class:     'AppBundle\TwitterClient'
8          autowire: true
```

The autowiring subsystem detects that the `rot13_transformer` service implements the `TransformerInterface` and injects it automatically. Even when using interfaces (and you should), building the service graph and refactoring the project is easier than with standard definitions.

Dealing with Multiple Implementations of the Same Type

Last but not least, the autowiring feature allows to specify the default implementation of a given type. Let's introduce a new implementation of the `TransformerInterface` returning the result of the ROT13 transformation uppercased:

Listing 38-10
```
1  // src/AppBundle/UppercaseRot13Transformer.php
2  namespace AppBundle;
3
4  class UppercaseTransformer implements TransformerInterface
5  {
6      private $transformer;
7
8      public function __construct(TransformerInterface $transformer)
9      {
10         $this->transformer = $transformer;
11     }
12
13     public function transform($value)
14     {
15         return strtoupper($this->transformer->transform($value));
16     }
17 }
```

This class is intended to decorate the any transformer and return its value uppercased.

We can now refactor the controller to add another endpoint leveraging this new transformer:

Listing 38-11
```
1  // src/AppBundle/Controller/DefaultController.php
2  namespace AppBundle\Controller;
3
```

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\BadRequestHttpException;

class DefaultController extends Controller
{
    /**
     * @Route("/tweet")
     * @Method("POST")
     */
    public function tweetAction(Request $request)
    {
        return $this->tweet($request, 'twitter_client');
    }

    /**
     * @Route("/tweet-uppercase")
     * @Method("POST")
     */
    public function tweetUppercaseAction(Request $request)
    {
        return $this->tweet($request, 'uppercase_twitter_client');
    }

    private function tweet(Request $request, $service)
    {
        $user = $request->request->get('user');
        $key = $request->request->get('key');
        $status = $request->request->get('status');

        if (!$user || !$key || !$status) {
            throw new BadRequestHttpException();
        }

        $this->get($service)->tweet($user, $key, $status);

        return new Response('OK');
    }
}
```

The last step is to update service definitions to register this new implementation and a Twitter client using it:

Listing 38-12

```yaml
# app/config/services.yml
services:
    rot13_transformer:
        class:            AppBundle\Rot13Transformer
        autowiring_types: AppBundle\TransformerInterface

    twitter_client:
        class:     AppBundle\TwitterClient
        autowire: true

    uppercase_rot13_transformer:
        class:     AppBundle\UppercaseRot13Transformer
```

```
13          autowire: true
14
15      uppercase_twitter_client:
16          class:      AppBundle\TwitterClient
17          arguments: ['@uppercase_rot13_transformer']
```

This deserves some explanations. You now have two services implementing the `TransformerInterface`. The autowiring subsystem cannot guess which one to use which leads to errors like this:

Listing 38-13
```
1 [Symfony\Component\DependencyInjection\Exception\RuntimeException]
2 Unable to autowire argument of type "AppBundle\TransformerInterface" for the service
  "twitter_client".
```

Fortunately, the `autowiring_types` key is here to specify which implementation to use by default. This key can take a list of types if necessary.

Thanks to this setting, the `rot13_transformer` service is automatically injected as an argument of the `uppercase_rot13_transformer` and `twitter_client` services. For the `uppercase_twitter_client`, we use a standard service definition to inject the specific `uppercase_rot13_transformer` service.

As for other RAD features such as the FrameworkBundle controller or annotations, keep in mind to not use autowiring in public bundles nor in large projects with complex maintenance needs.

Chapter 39

How to Inject Instances into the Container

When using the container in your application, you sometimes need to inject an instance instead of configuring the container to create a new instance.

For instance, if you're using the *HttpKernel* component with the DependencyInjection component, then the `kernel` service is injected into the container from within the `Kernel` class:

Listing 39-1

```
// ...
abstract class Kernel implements KernelInterface, TerminableInterface
{
    // ...
    protected function initializeContainer()
    {
        // ...
        $this->container->set('kernel', $this);

        // ...
    }
}
```

The `kernel` service is called a synthetic service. This service has to be configured in the container, so the container knows the service does exist during compilation (otherwise, services depending on this `kernel` service will get a "service does not exist" error).

In order to do so, you have to use *`Definition::setSynthetic()`*[1]:

Listing 39-2

```
use Symfony\Component\DependencyInjection\Definition;

// synthetic services don't specify a class
$kernelDefinition = new Definition();
$kernelDefinition->setSynthetic(true);

$container->setDefinition('your_service', $kernelDefinition);
```

Now, you can inject the instance in the container using *`Container::set()`*[2]:

1. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Definition.html#method_setSynthetic
2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Container.html#method_set

Listing 39-3

```php
$yourService = new YourObject();
$container->set('your_service', $yourService);
```

`$container->get('your_service')` will now return the same instance as `$yourService`.

Chapter 40

Compiling the Container

The service container can be compiled for various reasons. These reasons include checking for any potential issues such as circular references and making the container more efficient by resolving parameters and removing unused services. Also, certain features - like using *parent services* - require the container to be compiled.

It is compiled by running:

Listing 40-1
```
$container->compile();
```

The compile method uses *Compiler Passes* for the compilation. The DependencyInjection component comes with several passes which are automatically registered for compilation. For example the *CheckDefinitionValidityPass*[1] checks for various potential issues with the definitions that have been set in the container. After this and several other passes that check the container's validity, further compiler passes are used to optimize the configuration before it is cached. For example, private services and abstract services are removed and aliases are resolved.

Managing Configuration with Extensions

As well as loading configuration directly into the container as shown in *The DependencyInjection Component*, you can manage it by registering extensions with the container. The first step in the compilation process is to load configuration from any extension classes registered with the container. Unlike the configuration loaded directly, they are only processed when the container is compiled. If your application is modular then extensions allow each module to register and manage their own service configuration.

The extensions must implement *ExtensionInterface*[2] and can be registered with the container with:

Listing 40-2
```
$container->registerExtension($extension);
```

The main work of the extension is done in the `load` method. In the `load` method you can load configuration from one or more configuration files as well as manipulate the container definitions using the methods shown in *Working with Container Service Definitions*.

1. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Compiler/CheckDefinitionValidityPass.html
2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html

The `load` method is passed a fresh container to set up, which is then merged afterwards into the container it is registered with. This allows you to have several extensions managing container definitions independently. The extensions do not add to the containers configuration when they are added but are processed when the container's `compile` method is called.

A very simple extension may just load configuration files into the container:

Listing 40-3

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\DependencyInjection\Extension\ExtensionInterface;
use Symfony\Component\Config\FileLocator;

class AcmeDemoExtension implements ExtensionInterface
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader(
            $container,
            new FileLocator(__DIR__.'/../Resources/config')
        );
        $loader->load('services.xml');
    }

    // ...
}
```

This does not gain very much compared to loading the file directly into the overall container being built. It just allows the files to be split up amongst the modules/bundles. Being able to affect the configuration of a module from configuration files outside of the module/bundle is needed to make a complex application configurable. This can be done by specifying sections of config files loaded directly into the container as being for a particular extension. These sections on the config will not be processed directly by the container but by the relevant Extension.

The Extension must specify a `getAlias` method to implement the interface:

Listing 40-4

```
// ...

class AcmeDemoExtension implements ExtensionInterface
{
    // ...

    public function getAlias()
    {
        return 'acme_demo';
    }
}
```

For YAML configuration files specifying the alias for the extension as a key will mean that those values are passed to the Extension's `load` method:

Listing 40-5

```
# ...
acme_demo:
    foo: fooValue
    bar: barValue
```

If this file is loaded into the configuration then the values in it are only processed when the container is compiled at which point the Extensions are loaded:

Listing 40-6

```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2  use Symfony\Component\Config\FileLocator;
3  use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
4
5  $container = new ContainerBuilder();
6  $container->registerExtension(new AcmeDemoExtension);
7
8  $loader = new YamlFileLoader($container, new FileLocator(__DIR__));
9  $loader->load('config.yml');
10
11 // ...
12 $container->compile();
```

When loading a config file that uses an extension alias as a key, the extension must already have been registered with the container builder or an exception will be thrown.

The values from those sections of the config files are passed into the first argument of the `load` method of the extension:

Listing 40-7

```
1  public function load(array $configs, ContainerBuilder $container)
2  {
3      $foo = $configs[0]['foo']; //fooValue
4      $bar = $configs[0]['bar']; //barValue
5  }
```

The `$configs` argument is an array containing each different config file that was loaded into the container. You are only loading a single config file in the above example but it will still be within an array. The array will look like this:

Listing 40-8

```
1  array(
2      array(
3          'foo' => 'fooValue',
4          'bar' => 'barValue',
5      ),
6  )
```

Whilst you can manually manage merging the different files, it is much better to use *the Config component* to merge and validate the config values. Using the configuration processing you could access the config value this way:

Listing 40-9

```
1  use Symfony\Component\Config\Definition\Processor;
2  // ...
3
4  public function load(array $configs, ContainerBuilder $container)
5  {
6      $configuration = new Configuration();
7      $processor = new Processor();
8      $config = $processor->processConfiguration($configuration, $configs);
9
10     $foo = $config['foo']; //fooValue
11     $bar = $config['bar']; //barValue
12
```

```
13     // ...
14 }
```

There are a further two methods you must implement. One to return the XML namespace so that the relevant parts of an XML config file are passed to the extension. The other to specify the base path to XSD files to validate the XML configuration:

Listing 40-10
```
1 public function getXsdValidationBasePath()
2 {
3     return __DIR__.'/../Resources/config/';
4 }
5
6 public function getNamespace()
7 {
8     return 'http://www.example.com/symfony/schema/';
9 }
```

XSD validation is optional, returning `false` from the `getXsdValidationBasePath` method will disable it.

The XML version of the config would then look like this:

Listing 40-11
```
1  <?xml version="1.0" ?>
2  <container xmlns="http://symfony.com/schema/dic/services"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:acme_demo="http://www.example.com/symfony/schema/"
5      xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/
6  symfony/schema/hello-1.0.xsd">
7
8      <acme_demo:config>
9          <acme_demo:foo>fooValue</acme_hello:foo>
10         <acme_demo:bar>barValue</acme_demo:bar>
11     </acme_demo:config>
   </container>
```

In the Symfony full-stack Framework there is a base Extension class which implements these methods as well as a shortcut method for processing the configuration. See *How to Load Service Configuration inside a Bundle* for more details.

The processed config value can now be added as container parameters as if it were listed in a `parameters` section of the config file but with the additional benefit of merging multiple files and validation of the configuration:

Listing 40-12
```
1 public function load(array $configs, ContainerBuilder $container)
2 {
3     $configuration = new Configuration();
4     $processor = new Processor();
5     $config = $processor->processConfiguration($configuration, $configs);
6
7     $container->setParameter('acme_demo.FOO', $config['foo']);
8
```

```
9     // ...
10 }
```

More complex configuration requirements can be catered for in the Extension classes. For example, you may choose to load a main service configuration file but also load a secondary one only if a certain parameter is set:

Listing 40-13

```
1  public function load(array $configs, ContainerBuilder $container)
2  {
3      $configuration = new Configuration();
4      $processor = new Processor();
5      $config = $processor->processConfiguration($configuration, $configs);
6
7      $loader = new XmlFileLoader(
8          $container,
9          new FileLocator(__DIR__.'/../Resources/config')
10     );
11     $loader->load('services.xml');
12
13     if ($config['advanced']) {
14         $loader->load('advanced.xml');
15     }
16 }
```

Just registering an extension with the container is not enough to get it included in the processed extensions when the container is compiled. Loading config which uses the extension's alias as a key as in the above examples will ensure it is loaded. The container builder can also be told to load it with its *loadFromExtension()*[3] method:

Listing 40-14

```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2
3  $container = new ContainerBuilder();
4  $extension = new AcmeDemoExtension();
5  $container->registerExtension($extension);
6  $container->loadFromExtension($extension->getAlias());
7  $container->compile();
```

If you need to manipulate the configuration loaded by an extension then you cannot do it from another extension as it uses a fresh container. You should instead use a compiler pass which works with the full container after the extensions have been processed.

Prepending Configuration Passed to the Extension

An Extension can prepend the configuration of any Bundle before the `load()` method is called by implementing *PrependExtensionInterface*[4]:

Listing 40-15

3. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerBuilder.html#method_loadFromExtension
4. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/PrependExtensionInterface.html

```php
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
// ...

class AcmeDemoExtension implements ExtensionInterface, PrependExtensionInterface
{
    // ...

    public function prepend()
    {
        // ...

        $container->prependExtensionConfig($name, $config);

        // ...
    }
}
```

For more details, see *How to Simplify Configuration of multiple Bundles*, which is specific to the Symfony Framework, but contains more details about this feature.

Execute Code During Compilation

You can also execute custom code during compilation by writing your own compiler pass. By implementing *`CompilerPassInterface`*[5] in your extension, the added `process()` method will be called during compilation:

Listing 40-16

```php
// ...
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;

class AcmeDemoExtension implements ExtensionInterface, CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
       // ... do something during the compilation
    }

    // ...
}
```

As `process()` is called *after* all extensions are loaded, it allows you to edit service definitions of other extensions as well as retrieving information about service definitions.

The container's parameters and definitions can be manipulated using the methods described in *Working with Container Service Definitions*.

Please note that the `process()` method in the extension class is called during the optimization step. You can read the next section if you need to edit the container during another step.

5. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Compiler/CompilerPassInterface.html

As a rule, only work with services definition in a compiler pass and do not create service instances. In practice, this means using the methods `has()`, `findDefinition()`, `getDefinition()`, `setDefinition()`, etc. instead of `get()`, `set()`, etc.

Make sure your compiler pass does not require services to exist. Abort the method call if some required service is not available.

A common use-case of compiler passes is to search for all service definitions that have a certain tag in order to process dynamically plug each into some other service. See the section on service tags for an example.

Creating Separate Compiler Passes

Sometimes, you need to do more than one thing during compilation, want to use compiler passes without an extension or you need to execute some code at another step in the compilation process. In these cases, you can create a new class implementing the `CompilerPassInterface`:

Listing 40-17
```
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class CustomPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
       // ... do something during the compilation
    }
}
```

You then need to register your custom pass with the container:

Listing 40-18
```
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->addCompilerPass(new CustomPass());
```

Compiler passes are registered differently if you are using the full-stack framework, see *How to Work with Compiler Passes in Bundles* for more details.

Controlling the Pass Ordering

The default compiler passes are grouped into optimization passes and removal passes. The optimization passes run first and include tasks such as resolving references within the definitions. The removal passes perform tasks such as removing private aliases and unused services. When registering compiler passes using `addCompilerPass()`, you can configure when your compiler pass is run. By default, they are run before the optimization passes.

You can use the following constants to determine when your pass is executed:

- `PassConfig::TYPE_BEFORE_OPTIMIZATION`
- `PassConfig::TYPE_OPTIMIZE`
- `PassConfig::TYPE_BEFORE_REMOVING`

- PassConfig::TYPE_REMOVE
- PassConfig::TYPE_AFTER_REMOVING

For example, to run your custom pass after the default removal passes have been run, use:

Listing 40-19
```
// ...
$container->addCompilerPass(
    new CustomPass(),
    PassConfig::TYPE_AFTER_REMOVING
);
```

Dumping the Configuration for Performance

Using configuration files to manage the service container can be much easier to understand than using PHP once there are a lot of services. This ease comes at a price though when it comes to performance as the config files need to be parsed and the PHP configuration built from them. The compilation process makes the container more efficient but it takes time to run. You can have the best of both worlds though by using configuration files and then dumping and caching the resulting configuration. The `PhpDumper` makes dumping the compiled container easy:

Listing 40-20
```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Dumper\PhpDumper;

$file = __DIR__ .'/cache/container.php';

if (file_exists($file)) {
    require_once $file;
    $container = new ProjectServiceContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    $dumper = new PhpDumper($container);
    file_put_contents($file, $dumper->dump());
}
```

`ProjectServiceContainer` is the default name given to the dumped container class, you can change this though this with the `class` option when you dump it:

Listing 40-21
```
// ...
$file = __DIR__ .'/cache/container.php';

if (file_exists($file)) {
    require_once $file;
    $container = new MyCachedContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    $dumper = new PhpDumper($container);
    file_put_contents(
        $file,
        $dumper->dump(array('class' => 'MyCachedContainer'))
```

```
16     );
17 }
```

You will now get the speed of the PHP configured container with the ease of using configuration files. Additionally dumping the container in this way further optimizes how the services are created by the container.

In the above example you will need to delete the cached container file whenever you make any changes. Adding a check for a variable that determines if you are in debug mode allows you to keep the speed of the cached container in production but getting an up to date configuration whilst developing your application:

Listing 40-22

```
// ...

// based on something in your project
$isDebug = ...;

$file = __DIR__ .'/cache/container.php';

if (!$isDebug && file_exists($file)) {
    require_once $file;
    $container = new MyCachedContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    if (!$isDebug) {
        $dumper = new PhpDumper($container);
        file_put_contents(
            $file,
            $dumper->dump(array('class' => 'MyCachedContainer'))
        );
    }
}
```

This could be further improved by only recompiling the container in debug mode when changes have been made to its configuration rather than on every request. This can be done by caching the resource files used to configure the container in the way described in "*Caching based on Resources*" in the config component documentation.

You do not need to work out which files to cache as the container builder keeps track of all the resources used to configure it, not just the configuration files but the extension classes and compiler passes as well. This means that any changes to any of these files will invalidate the cache and trigger the container being rebuilt. You just need to ask the container for these resources and use them as metadata for the cache:

Listing 40-23

```
// ...

// based on something in your project
$isDebug = ...;

$file = __DIR__ .'/cache/container.php';
$containerConfigCache = new ConfigCache($file, $isDebug);

if (!$containerConfigCache->isFresh()) {
    $containerBuilder = new ContainerBuilder();
    // ...
```

```
12     $containerBuilder->compile();
13
14     $dumper = new PhpDumper($containerBuilder);
15     $containerConfigCache->write(
16         $dumper->dump(array('class' => 'MyCachedContainer')),
17         $containerBuilder->getResources()
18     );
19 }
20
21 require_once $file;
22 $container = new MyCachedContainer();
```

Now the cached dumped container is used regardless of whether debug mode is on or not. The difference is that the `ConfigCache` is set to debug mode with its second constructor argument. When the cache is not in debug mode the cached container will always be used if it exists. In debug mode, an additional metadata file is written with the timestamps of all the resource files. These are then checked to see if the files have changed, if they have the cache will be considered stale.

In the full-stack framework the compilation and caching of the container is taken care of for you.

Chapter 41

Working with Tagged Services

Tags are a generic string (along with some options) that can be applied to any service. By themselves, tags don't actually alter the functionality of your services in any way. But if you choose to, you can ask a container builder for a list of all services that were tagged with some specific tag. This is useful in compiler passes where you can find these services and use or modify them in some specific way.

For example, if you are using Swift Mailer you might imagine that you want to implement a "transport chain", which is a collection of classes implementing `\Swift_Transport`. Using the chain, you'll want Swift Mailer to try several ways of transporting the message until one succeeds.

To begin with, define the `TransportChain` class:

Listing 41-1

```
1  class TransportChain
2  {
3      private $transports;
4
5      public function __construct()
6      {
7          $this->transports = array();
8      }
9
10     public function addTransport(\Swift_Transport $transport)
11     {
12         $this->transports[] = $transport;
13     }
14 }
```

Then, define the chain as a service:

Listing 41-2

```
1 services:
2     acme_mailer.transport_chain:
3         class: TransportChain
```

Define Services with a Custom Tag

Now you might want several of the `\Swift_Transport` classes to be instantiated and added to the chain automatically using the `addTransport()` method. For example you may add the following transports as services:

Listing 41-3

```
services:
    acme_mailer.transport.smtp:
        class: \Swift_SmtpTransport
        arguments:
            - '%mailer_host%'
        tags:
            -  { name: acme_mailer.transport }
    acme_mailer.transport.sendmail:
        class: \Swift_SendmailTransport
        tags:
            -  { name: acme_mailer.transport }
```

Notice that each was given a tag named `acme_mailer.transport`. This is the custom tag that you'll use in your compiler pass. The compiler pass is what makes this tag "mean" something.

Create a Compiler Pass

You can now use a compiler pass to ask the container for any services with the `acme_mailer.transport` tag:

Listing 41-4

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (!$container->has('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->findDefinition(
            'acme_mailer.transport_chain'
        );

        $taggedServices = $container->findTaggedServiceIds(
            'acme_mailer.transport'
        );
        foreach ($taggedServices as $id => $tags) {
            $definition->addMethodCall(
                'addTransport',
                array(new Reference($id))
            );
        }
    }
}
```

The `process()` method checks for the existence of the `acme_mailer.transport_chain` service, then looks for all services tagged `acme_mailer.transport`. It adds to the definition of the

acme_mailer.transport_chain service a call to addTransport() for each acme_mailer.transport service it has found. The first argument of each of these calls will be the mailer transport service itself.

Register the Pass with the Container

You also need to register the pass with the container, it will then be run when the container is compiled:

Listing 41-5
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->addCompilerPass(new TransportCompilerPass());
```

Compiler passes are registered differently if you are using the full-stack framework. See *How to Work with Compiler Passes in Bundles* for more details.

When implementing the `CompilerPassInterface` in a service extension, you do not need to register it. See the components documentation for more information.

Adding Additional Attributes on Tags

Sometimes you need additional information about each service that's tagged with your tag. For example, you might want to add an alias to each member of the transport chain.

To begin with, change the `TransportChain` class:

Listing 41-6
```php
class TransportChain
{
    private $transports;

    public function __construct()
    {
        $this->transports = array();
    }

    public function addTransport(\Swift_Transport $transport, $alias)
    {
        $this->transports[$alias] = $transport;
    }

    public function getTransport($alias)
    {
        if (array_key_exists($alias, $this->transports)) {
            return $this->transports[$alias];
        }
    }
}
```

As you can see, when `addTransport` is called, it takes not only a `Swift_Transport` object, but also a string alias for that transport. So, how can you allow each tagged transport service to also supply an alias?

To answer this, change the service declaration:

Listing 41-7

```yaml
services:
    acme_mailer.transport.smtp:
        class: \Swift_SmtpTransport
        arguments:
            - '%mailer_host%'
        tags:
            - { name: acme_mailer.transport, alias: foo }
    acme_mailer.transport.sendmail:
        class: \Swift_SendmailTransport
        tags:
            - { name: acme_mailer.transport, alias: bar }
```

Notice that you've added a generic `alias` key to the tag. To actually use this, update the compiler:

Listing 41-8

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (!$container->hasDefinition('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->getDefinition(
            'acme_mailer.transport_chain'
        );

        $taggedServices = $container->findTaggedServiceIds(
            'acme_mailer.transport'
        );
        foreach ($taggedServices as $id => $tags) {
            foreach ($tags as $attributes) {
                $definition->addMethodCall(
                    'addTransport',
                    array(new Reference($id), $attributes["alias"])
                );
            }
        }
    }
}
```

The double loop may be confusing. This is because a service can have more than one tag. You tag a service twice or more with the `acme_mailer.transport` tag. The second foreach loop iterates over the `acme_mailer.transport` tags set for the current service and gives you the attributes.

Chapter 42

Using a Factory to Create Services

Symfony's Service Container provides a powerful way of controlling the creation of objects, allowing you to specify arguments passed to the constructor as well as calling methods and setting parameters. Sometimes, however, this will not provide you with everything you need to construct your objects. For this situation, you can use a factory to create the object and tell the service container to call a method on the factory rather than directly instantiating the class.

Suppose you have a factory that configures and returns a new `NewsletterManager` object:

Listing 42-1

```
class NewsletterManagerFactory
{
    public static function createNewsletterManager()
    {
        $newsletterManager = new NewsletterManager();

        // ...

        return $newsletterManager;
    }
}
```

To make the `NewsletterManager` object available as a service, you can configure the service container to use the `NewsletterManagerFactory::createNewsletterManager()` factory method:

Listing 42-2

```
services:
    newsletter_manager:
        class:   NewsletterManager
        factory: [NewsletterManagerFactory, createNewsletterManager]
```

When using a factory to create services, the value chosen for the `class` option has no effect on the resulting service. The actual class name only depends on the object that is returned by the factory. However, the configured class name may be used by compiler passes and therefore should be set to a sensible value.

Now, the method will be called statically. If the factory class itself should be instantiated and the resulting object's method called, configure the factory itself as a service. In this case, the method (e.g. get) should be changed to be non-static.

Listing 42-3

```
1 services:
2     newsletter_manager.factory:
3         class: NewsletterManagerFactory
4     newsletter_manager:
5         class:   NewsletterManager
6         factory: ["@newsletter_manager.factory", createNewsletterManager]
```

Passing Arguments to the Factory Method

If you need to pass arguments to the factory method, you can use the `arguments` options inside the service container. For example, suppose the `createNewsletterManager` method in the previous example takes the `templating` service as an argument:

Listing 42-4

```
1 services:
2     newsletter_manager.factory:
3         class: NewsletterManagerFactory
4 
5     newsletter_manager:
6         class:   NewsletterManager
7         factory: ["@newsletter_manager.factory", createNewsletterManager]
8         arguments:
9             - '@templating'
```

Chapter 43

Configuring Services with a Service Configurator

The Service Configurator is a feature of the Dependency Injection Container that allows you to use a callable to configure a service after its instantiation.

You can specify a method in another service, a PHP function or a static method in a class. The service instance is passed to the callable, allowing the configurator to do whatever it needs to configure the service after its creation.

A Service Configurator can be used, for example, when you have a service that requires complex setup based on configuration settings coming from different sources/services. Using an external configurator, you can maintain the service implementation cleanly and keep it decoupled from the other objects that provide the configuration needed.

Another interesting use case is when you have multiple objects that share a common configuration or that should be configured in a similar way at runtime.

For example, suppose you have an application where you send different types of emails to users. Emails are passed through different formatters that could be enabled or not depending on some dynamic application settings. You start defining a `NewsletterManager` class like this:

Listing 43-1

```
class NewsletterManager implements EmailFormatterAwareInterface
{
    protected $mailer;
    protected $enabledFormatters;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEnabledFormatters(array $enabledFormatters)
    {
        $this->enabledFormatters = $enabledFormatters;
    }

```

```
16     // ...
17 }
```

and also a `GreetingCardManager` class:

Listing 43-2

```
1  class GreetingCardManager implements EmailFormatterAwareInterface
2  {
3      protected $mailer;
4      protected $enabledFormatters;
5
6      public function setMailer(Mailer $mailer)
7      {
8          $this->mailer = $mailer;
9      }
10
11     public function setEnabledFormatters(array $enabledFormatters)
12     {
13         $this->enabledFormatters = $enabledFormatters;
14     }
15
16     // ...
17 }
```

As mentioned before, the goal is to set the formatters at runtime depending on application settings. To do this, you also have an `EmailFormatterManager` class which is responsible for loading and validating formatters enabled in the application:

Listing 43-3

```
1  class EmailFormatterManager
2  {
3      protected $enabledFormatters;
4
5      public function loadFormatters()
6      {
7          // code to configure which formatters to use
8          $enabledFormatters = array(...);
9          // ...
10
11         $this->enabledFormatters = $enabledFormatters;
12     }
13
14     public function getEnabledFormatters()
15     {
16         return $this->enabledFormatters;
17     }
18
19     // ...
20 }
```

If your goal is to avoid having to couple `NewsletterManager` and `GreetingCardManager` with `EmailFormatterManager`, then you might want to create a configurator class to configure these instances:

Listing 43-4

```
1  class EmailConfigurator
2  {
3      private $formatterManager;
4
5      public function __construct(EmailFormatterManager $formatterManager)
```

```
6       {
7           $this->formatterManager = $formatterManager;
8       }
9
10      public function configure(EmailFormatterAwareInterface $emailManager)
11      {
12          $emailManager->setEnabledFormatters(
13              $this->formatterManager->getEnabledFormatters()
14          );
15      }
16
17      // ...
18  }
```

The EmailConfigurator's job is to inject the enabled formatters into NewsletterManager and GreetingCardManager because they are not aware of where the enabled formatters come from. On the other hand, the EmailFormatterManager holds the knowledge about the enabled formatters and how to load them, keeping the single responsibility principle.

Configurator Service Config

The service config for the above classes would look something like this:

Listing 43-5

```
1   services:
2       my_mailer:
3           # ...
4
5       email_formatter_manager:
6           class:     EmailFormatterManager
7           # ...
8
9       email_configurator:
10          class:     EmailConfigurator
11          arguments: ['@email_formatter_manager']
12          # ...
13
14      newsletter_manager:
15          class:     NewsletterManager
16          calls:
17              - [setMailer, ['@my_mailer']]
18          configurator: ['@email_configurator', configure]
19
20      greeting_card_manager:
21          class:     GreetingCardManager
22          calls:
23              - [setMailer, ['@my_mailer']]
24          configurator: ['@email_configurator', configure]
```

Chapter 44

Managing Common Dependencies with Parent Services

As you add more functionality to your application, you may well start to have related classes that share some of the same dependencies. For example, you may have a Newsletter Manager which uses setter injection to set its dependencies:

Listing 44-1

```
class NewsletterManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }

    // ...
}
```

and also a Greeting Card class which shares the same dependencies:

Listing 44-2

```
class GreetingCardManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
```

```
 9      }
10
11      public function setEmailFormatter(EmailFormatter $emailFormatter)
12      {
13          $this->emailFormatter = $emailFormatter;
14      }
15
16      // ...
17  }
```

The service config for these classes would look something like this:

Listing 44-3

```
 1  services:
 2      my_mailer:
 3          # ...
 4
 5      my_email_formatter:
 6          # ...
 7
 8      newsletter_manager:
 9          class: NewsletterManager
10          calls:
11              - [setMailer, ['@my_mailer']]
12              - [setEmailFormatter, ['@my_email_formatter']]
13
14      greeting_card_manager:
15          class: 'GreetingCardManager'
16          calls:
17              - [setMailer, ['@my_mailer']]
18              - [setEmailFormatter, ['@my_email_formatter']]
```

There is a lot of repetition in both the classes and the configuration. This means that if you changed, for example, the `Mailer` of `EmailFormatter` classes to be injected via the constructor, you would need to update the config in two places. Likewise if you needed to make changes to the setter methods you would need to do this in both classes. The typical way to deal with the common methods of these related classes would be to extract them to a super class:

Listing 44-4

```
 1  abstract class MailManager
 2  {
 3      protected $mailer;
 4      protected $emailFormatter;
 5
 6      public function setMailer(Mailer $mailer)
 7      {
 8          $this->mailer = $mailer;
 9      }
10
11      public function setEmailFormatter(EmailFormatter $emailFormatter)
12      {
13          $this->emailFormatter = $emailFormatter;
14      }
15
16      // ...
17  }
```

The `NewsletterManager` and `GreetingCardManager` can then extend this super class:

Listing 44-5

```
class NewsletterManager extends MailManager
{
    // ...
}
```

and:

Listing 44-6

```
class GreetingCardManager extends MailManager
{
    // ...
}
```

In a similar fashion, the Symfony service container also supports extending services in the configuration so you can also reduce the repetition by specifying a parent for a service.

Listing 44-7

```
1  # ...
2  services:
3      # ...
4      mail_manager:
5          abstract:  true
6          calls:
7              - [setMailer, ['@my_mailer']]
8              - [setEmailFormatter, ['@my_email_formatter']]
9
10     newsletter_manager:
11         class:  "NewsletterManager"
12         parent: mail_manager
13
14     greeting_card_manager:
15         class:  "GreetingCardManager"
16         parent: mail_manager
```

In this context, having a `parent` service implies that the arguments and method calls of the parent service should be used for the child services. Specifically, the setter methods defined for the parent service will be called when the child services are instantiated.

If you remove the `parent` config key, the services will still be instantiated and they will still of course extend the `MailManager` class. The difference is that omitting the `parent` config key will mean that the `calls` defined on the `mail_manager` service will not be executed when the child services are instantiated.

The `abstract` and `tags` attributes are always taken from the child service.

The parent service is abstract as it should not be directly retrieved from the container or passed into another service. It exists merely as a "template" that other services can use. This is why it can have no `class` configured which would cause an exception to be raised for a non-abstract service.

In order for parent dependencies to resolve, the `ContainerBuilder` must first be compiled. See *Compiling the Container* for more details.

In the examples shown, the classes sharing the same configuration also extend from the same parent class in PHP. This isn't necessary at all. You can just extract common parts of similar service definitions into a parent service without also extending a parent class in PHP.

Overriding Parent Dependencies

There may be times where you want to override what class is passed in for a dependency of one child service only. Fortunately, by adding the method call config for the child service, the dependencies set by the parent class will be overridden. So if you needed to pass a different dependency just to the `NewsletterManager` class, the config would look like this:

Listing 44-8

```
# ...
services:
    # ...
    my_alternative_mailer:
        # ...

    mail_manager:
        abstract: true
        calls:
            - [setMailer, ['@my_mailer']]
            - [setEmailFormatter, ['@my_email_formatter']]

    newsletter_manager:
        class:  'NewsletterManager'
        parent: mail_manager
        calls:
            - [setMailer, ['@my_alternative_mailer']]

    greeting_card_manager:
        class:  'GreetingCardManager'
        parent: mail_manager
```

The `GreetingCardManager` will receive the same dependencies as before, but the `NewsletterManager` will be passed the `my_alternative_mailer` instead of the `my_mailer` service.

You can't override method calls. When you defined new method calls in the child service, it'll be added to the current set of configured method calls. This means it works perfectly when the setter overrides the current property, but it doesn't work as expected when the setter appends it to the existing data (e.g. an `addFilters()` method). In those cases, the only solution is to *not* extend the parent service and configuring the service just like you did before knowing this feature.

Chapter 45
Advanced Container Configuration

Marking Services as Public / Private

When defining services, you'll usually want to be able to access these definitions within your application code. These services are called `public`. For example, the `doctrine` service registered with the container when using the DoctrineBundle is a public service. This means that you can fetch it from the container using the `get()` method:

Listing 45-1
```
$doctrine = $container->get('doctrine');
```

In some cases, a service *only* exists to be injected into another service and is *not* intended to be fetched directly from the container as shown above.

In these cases, to get a minor performance boost, you can set the service to be *not* public (i.e. private):

Listing 45-2
```
1 services:
2    foo:
3      class: Example\Foo
4      public: false
```

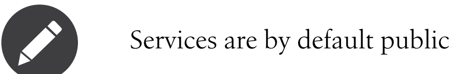
What makes private services special is that, if they are only injected once, they are converted from services to inlined instantiations (e.g. `new PrivateThing()`). This increases the container's performance.

Now that the service is private, you *should not* fetch the service directly from the container:

Listing 45-3
```
$container->get('foo');
```

This *may or may not work*, depending on if the service could be inlined. Simply said: A service can be marked as private if you do not want to access it directly from your code.

However, if a service has been marked as private, you can still alias it (see below) to access this service (via the alias).



Services are by default public.

Aliasing

You may sometimes want to use shortcuts to access some services. You can do so by aliasing them and, furthermore, you can even alias non-public services.

Listing 45-4
```
1 services:
2     foo:
3       class: Example\Foo
4     bar:
5       alias: foo
```

This means that when using the container directly, you can access the `foo` service by asking for the `bar` service like this:

Listing 45-5
```
$container->get('bar'); // Would return the foo service
```

In YAML, you can also use a shortcut to alias a service:

Listing 45-6
```
1 services:
2     foo:
3       class: Example\Foo
4     bar: '@foo'
```

Decorating Services

When overriding an existing definition, the old service is lost:

Listing 45-7
```
1 $container->register('foo', 'FooService');
2
3 // this is going to replace the old definition with the new one
4 // old definition is lost
5 $container->register('foo', 'CustomFooService');
```

Most of the time, that's exactly what you want to do. But sometimes, you might want to decorate the old one instead. In this case, the old service should be kept around to be able to reference it in the new one. This configuration replaces `foo` with a new one, but keeps a reference of the old one as `bar.inner`:

Listing 45-8
```
1 bar:
2   public: false
3   class: stdClass
4   decorates: foo
5   arguments: ["@bar.inner"]
```

Here is what's going on here: the `setDecoratedService()` method tells the container that the `bar` service should replace the `foo` service, renaming `foo` to `bar.inner`. By convention, the old `foo` service is going to be renamed `bar.inner`, so you can inject it into your new service.

The generated inner id is based on the id of the decorator service (`bar` here), not of the decorated service (`foo` here). This is mandatory to allow several decorators on the same service (they need to have different generated inner ids).

Most of the time, the decorator should be declared private, as you will not need to retrieve it as `bar` from the container. The visibility of the decorated `foo` service (which is an alias for `bar`) will still be the same as the original `foo` visibility.

You can change the inner service name if you want to:

Listing 45-9
```
1 bar:
2   class: stdClass
3   public: false
4   decorates: foo
5   decoration_inner_name: bar.wooz
6   arguments: ["@bar.wooz"]
```

If you want to apply more than one decorator to a service, you can control their order by configuring the priority of decoration, this can be any integer number (decorators with higher priorities will be applied first).

Listing 45-10
```
1  foo:
2      class: Foo
3
4  bar:
5      class: Bar
6      public: false
7      decorates: foo
8      decoration_priority: 5
9      arguments: ['@bar.inner']
10
11 baz:
12     class: Baz
13     public: false
14     decorates: foo
15     decoration_priority: 1
16     arguments: ['@baz.inner']
```

The generated code will be the following:

Listing 45-11
```
1 $this->services['foo'] = new Baz(new Bar(new Foo()));
```

Deprecating Services

Once you have decided to deprecate the use of a service (because it is outdated or you decided not to maintain it anymore), you can deprecate its definition:

Listing 45-12

```
1 acme.my_service:
2     class: ...
3     deprecated: The "%service_id%" service is deprecated since 2.8 and will be removed in
  3.0.
```

Now, every time this service is used, a deprecation warning is triggered, advising you to stop or to change your uses of that service.

The message is actually a message template, which replaces occurrences of the `%service_id%` placeholder by the service's id. You **must** have at least one occurrence of the `%service_id%` placeholder in your template.

The deprecation message is optional. If not set, Symfony will show this default message: `The "%service_id%" service is deprecated. You should stop using it, as it will soon be removed.`.

It is strongly recommended that you define a custom message because the default one is too generic. A good message informs when this service was deprecated, until when it will be maintained and the alternative services to use (if any).

For service decorators (see above), if the definition does not modify the deprecated status, it will inherit the status from the definition that is decorated.

The ability to "un-deprecate" a service is possible only when declaring the definition in PHP.

Chapter 46
Lazy Services

Why Lazy Services?

In some cases, you may want to inject a service that is a bit heavy to instantiate, but is not always used inside your object. For example, imagine you have a `NewsletterManager` and you inject a `mailer` service into it. Only a few methods on your `NewsletterManager` actually use the `mailer`, but even when you don't need it, a `mailer` service is always instantiated in order to construct your `NewsletterManager`.

Configuring lazy services is one answer to this. With a lazy service, a "proxy" of the `mailer` service is actually injected. It looks and acts just like the `mailer`, except that the `mailer` isn't actually instantiated until you interact with the proxy in some way.

Installation

In order to use the lazy service instantiation, you will first need to install the *ProxyManager bridge*[1]:

Listing 46-1
```
1 $ composer require symfony/proxy-manager-bridge
```

If you're using the full-stack framework, the proxy manager bridge is already included but the actual proxy manager needs to be included. So, run:

Listing 46-2
```
1 $ composer require ocramius/proxy-manager:~1.0
```

Afterwards compile your container and check to make sure that you get a proxy for your lazy services.

1. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/ProxyManager

Configuration

You can mark the service as `lazy` by manipulating its definition:

Listing 46-3
```
1 services:
2   foo:
3     class: Acme\Foo
4     lazy: true
```

You can then require the service from the container:

Listing 46-4
```
$service = $container->get('foo');
```

At this point the retrieved `$service` should be a virtual *proxy*[2] with the same signature of the class representing the service. You can also inject the service just like normal into other services. The object that's actually injected will be the proxy.

To check if your proxy works you can simply check the interface of the received object.

Listing 46-5
```
1 var_dump(class_implements($service));
```

If the class implements the `ProxyManager\Proxy\LazyLoadingInterface` your lazy loaded services are working.

If you don't install the *ProxyManager bridge*[3], the container will just skip over the `lazy` flag and simply instantiate the service as it would normally do.

The proxy gets initialized and the actual service is instantiated as soon as you interact in any way with this object.

Additional Resources

You can read more about how proxies are instantiated, generated and initialized in the *documentation of ProxyManager*[4].

2. https://en.wikipedia.org/wiki/Proxy_pattern
3. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/ProxyManager
4. https://github.com/Ocramius/ProxyManager/blob/master/docs/lazy-loading-value-holder.md

Chapter 47
Container Building Workflow

In the preceding pages of this section, there has been little to say about where the various files and classes should be located. This is because this depends on the application, library or framework in which you want to use the container. Looking at how the container is configured and built in the Symfony full-stack Framework will help you see how this all fits together, whether you are using the full-stack framework or looking to use the service container in another application.

The full-stack framework uses the HttpKernel component to manage the loading of the service container configuration from the application and bundles and also handles the compilation and caching. Even if you are not using HttpKernel, it should give you an idea of one way of organizing configuration in a modular application.

Working with a Cached Container

Before building it, the kernel checks to see if a cached version of the container exists. The HttpKernel has a debug setting and if this is false, the cached version is used if it exists. If debug is true then the kernel *checks to see if configuration is fresh* and if it is, the cached version of the container is used. If not then the container is built from the application-level configuration and the bundles's extension configuration.

Read Dumping the Configuration for Performance for more details.

Application-level Configuration

Application level config is loaded from the `app/config` directory. Multiple files are loaded which are then merged when the extensions are processed. This allows for different configuration for different environments e.g. dev, prod.

These files contain parameters and services that are loaded directly into the container as per Setting Up the Container with Configuration Files. They also contain configuration that is processed by extensions as per Managing Configuration with Extensions. These are considered to be bundle configuration since each bundle contains an Extension class.

Bundle-level Configuration with Extensions

By convention, each bundle contains an Extension class which is in the bundle's `DependencyInjection` directory. These are registered with the `ContainerBuilder` when the kernel is booted. When the `ContainerBuilder` is *compiled*, the application-level configuration relevant to the bundle's extension is passed to the Extension which also usually loads its own config file(s), typically from the bundle's `Resources/config` directory. The application-level config is usually processed with a *Configuration object* also stored in the bundle's `DependencyInjection` directory.

Compiler Passes to Allow Interaction between Bundles

Compiler passes are used to allow interaction between different bundles as they cannot affect each other's configuration in the extension classes. One of the main uses is to process tagged services, allowing bundles to register services to be picked up by other bundles, such as Monolog loggers, Twig extensions and Data Collectors for the Web Profiler. Compiler passes are usually placed in the bundle's `DependencyInjection/Compiler` directory.

Compilation and Caching

After the compilation process has loaded the services from the configuration, extensions and the compiler passes, it is dumped so that the cache can be used next time. The dumped version is then used during subsequent requests as it is more efficient.

Chapter 48
The DomCrawler Component

> The DomCrawler component eases DOM navigation for HTML and XML documents.

While possible, the DomCrawler component is not designed for manipulation of the DOM or re-dumping HTML/XML.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/dom-crawler` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/dom-crawler*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The *`Crawler`*[2] class provides methods to query and manipulate HTML and XML documents.

An instance of the Crawler represents a set of *`DOMElement`*[3] objects, which are basically nodes that you can traverse easily:

Listing 48-1

1. https://packagist.org/packages/symfony/dom-crawler
2. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html
3. http://php.net/manual/en/class.domelement.php

```
1  use Symfony\Component\DomCrawler\Crawler;
2
3  $html = <<<'HTML'
4  <!DOCTYPE html>
5  <html>
6      <body>
7          <p class="message">Hello World!</p>
8          <p>Hello Crawler!</p>
9      </body>
10 </html>
11 HTML;
12
13 $crawler = new Crawler($html);
14
15 foreach ($crawler as $domElement) {
16     var_dump($domElement->nodeName);
17 }
```

Specialized `Link`[4] and `Form`[5] classes are useful for interacting with html links and forms as you traverse through the HTML tree.

The DomCrawler will attempt to automatically fix your HTML to match the official specification. For example, if you nest a `<p>` tag inside another `<p>` tag, it will be moved to be a sibling of the parent tag. This is expected and is part of the HTML5 spec. But if you're getting unexpected behavior, this could be a cause. And while the DomCrawler isn't meant to dump content, you can see the "fixed" version of your HTML by dumping it.

Node Filtering

Using XPath expressions is really easy:

Listing 48-2
```
$crawler = $crawler->filterXPath('descendant-or-self::body/p');
```

`DOMXPath::query` is used internally to actually perform an XPath query.

Filtering is even easier if you have the CssSelector component installed. This allows you to use jQuery-like selectors to traverse:

Listing 48-3
```
$crawler = $crawler->filter('body > p');
```

Anonymous function can be used to filter with more complex criteria:

Listing 48-4
```
1  use Symfony\Component\DomCrawler\Crawler;
2  // ...
3
4  $crawler = $crawler
5      ->filter('body > p')
6      ->reduce(function (Crawler $node, $i) {
7          // filter every other node
```

4. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Link.html
5. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Form.html

```
8           return ($i % 2) == 0;
9       });
```

To remove a node the anonymous function must return false.

All filter methods return a new *Crawler*[6] instance with filtered content.

Both the *filterXPath()*[7] and *filter()*[8] methods work with XML namespaces, which can be either automatically discovered or registered explicitly.

Consider the XML below:

Listing 48-5

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <entry
3      xmlns="http://www.w3.org/2005/Atom"
4      xmlns:media="http://search.yahoo.com/mrss/"
5      xmlns:yt="http://gdata.youtube.com/schemas/2007"
6  >
7      <id>tag:youtube.com,2008:video:kgZRZmEc9j4</id>
8      <yt:accessControl action="comment" permission="allowed"/>
9      <yt:accessControl action="videoRespond" permission="moderated"/>
10     <media:group>
11         <media:title type="plain">Chordates - CrashCourse Biology #24</media:title>
12         <yt:aspectRatio>widescreen</yt:aspectRatio>
13     </media:group>
14 </entry>
```

This can be filtered with the `Crawler` without needing to register namespace aliases both with *filterXPath()*[9]:

Listing 48-6

```
$crawler = $crawler->filterXPath('//default:entry/media:group//yt:aspectRatio');
```

and *filter()*[10]:

Listing 48-7

```
use Symfony\Component\CssSelector\CssSelector;

CssSelector::disableHtmlExtension();
$crawler = $crawler->filter('default|entry media|group yt|aspectRatio');
```

The default namespace is registered with a prefix "default". It can be changed with the *setDefaultNamespacePrefix()*[11] method.

The default namespace is removed when loading the content if it's the only namespace in the document. It's done to simplify the xpath queries.

Namespaces can be explicitly registered with the *registerNamespace()*[12] method:

6. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html
7. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_filterXPath
8. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_filter
9. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_filterXPath
10. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_filter
11. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_setDefaultNamespacePrefix
12. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_registerNamespace

Listing 48-8

```
$crawler->registerNamespace('m', 'http://search.yahoo.com/mrss/');
$crawler = $crawler->filterXPath('//m:group//yt:aspectRatio');
```

To query XML with a CSS selector, the HTML extension needs to be disabled with *CssSelector::disableHtmlExtension*[13] to avoid converting the selector to lowercase.

Node Traversing

Access node by its position on the list:

Listing 48-9

```
$crawler->filter('body > p')->eq(0);
```

Get the first or last node of the current selection:

Listing 48-10

```
$crawler->filter('body > p')->first();
$crawler->filter('body > p')->last();
```

Get the nodes of the same level as the current selection:

Listing 48-11

```
$crawler->filter('body > p')->siblings();
```

Get the same level nodes after or before the current selection:

Listing 48-12

```
$crawler->filter('body > p')->nextAll();
$crawler->filter('body > p')->previousAll();
```

Get all the child or parent nodes:

Listing 48-13

```
$crawler->filter('body')->children();
$crawler->filter('body > p')->parents();
```

All the traversal methods return a new *Crawler*[14] instance.

Accessing Node Values

Access the node name (HTML tag name) of the first node of the current selection (eg. "p" or "div"):

Listing 48-14

```
// will return the node name (HTML tag name) of the first child element under <body>
$tag = $crawler->filterXPath('//body/*')->nodeName();
```

Access the value of the first node of the current selection:

Listing 48-15

```
$message = $crawler->filterXPath('//body/p')->text();
```

Access the attribute value of the first node of the current selection:

Listing 48-16

```
$class = $crawler->filterXPath('//body/p')->attr('class');
```

Extract attribute and/or node values from the list of nodes:

13. http://api.symfony.com/3.0/Symfony/Component/CssSelector/CssSelector.html#method_disableHtmlExtension
14. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html

Listing 48-17

```
$attributes = $crawler
    ->filterXpath('//body/p')
    ->extract(array('_text', 'class'))
;
```

Special attribute `_text` represents a node value.

Call an anonymous function on each node of the list:

Listing 48-18

```
1 use Symfony\Component\DomCrawler\Crawler;
2 // ...
3
4 $nodeValues = $crawler->filter('p')->each(function (Crawler $node, $i) {
5     return $node->text();
6 });
```

The anonymous function receives the node (as a Crawler) and the position as arguments. The result is an array of values returned by the anonymous function calls.

Adding the Content

The crawler supports multiple ways of adding the content:

Listing 48-19

```
 1 $crawler = new Crawler('<html><body /></html>');
 2
 3 $crawler->addHtmlContent('<html><body /></html>');
 4 $crawler->addXmlContent('<root><node /></root>');
 5
 6 $crawler->addContent('<html><body /></html>');
 7 $crawler->addContent('<root><node /></root>', 'text/xml');
 8
 9 $crawler->add('<html><body /></html>');
10 $crawler->add('<root><node /></root>');
```

When dealing with character sets other than ISO-8859-1, always add HTML content using the *addHtmlContent()*[15] method where you can specify the second parameter to be your target character set.

As the Crawler's implementation is based on the DOM extension, it is also able to interact with native *DOMDocument*[16], *DOMNodeList*[17] and *DOMNode*[18] objects:

Listing 48-20

```
1 $document = new \DOMDocument();
2 $document->loadXml('<root><node /><node /></root>');
3 $nodeList = $document->getElementsByTagName('node');
4 $node = $document->getElementsByTagName('node')->item(0);
5
6 $crawler->addDocument($document);
```

15. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_addHtmlContent
16. http://php.net/manual/en/class.domdocument.php
17. http://php.net/manual/en/class.domnodelist.php
18. http://php.net/manual/en/class.domnode.php

```
7  $crawler->addNodeList($nodeList);
8  $crawler->addNodes(array($node));
9  $crawler->addNode($node);
10 $crawler->add($document);
```

Manipulating and Dumping a `Crawler`

These methods on the `Crawler` are intended to initially populate your `Crawler` and aren't intended to be used to further manipulate a DOM (though this is possible). However, since the `Crawler` is a set of *DOMElement*[19] objects, you can use any method or property available on *DOMElement*[20], *DOMNode*[21] or *DOMDocument*[22]. For example, you could get the HTML of a `Crawler` with something like this:

Listing 48-21
```
1 $html = '';
2
3 foreach ($crawler as $domElement) {
4     $html .= $domElement->ownerDocument->saveHTML($domElement);
5 }
```

Or you can get the HTML of the first node using *html()*[23]:

Listing 48-22
```
$html = $crawler->html();
```

Links

To find a link by name (or a clickable image by its `alt` attribute), use the `selectLink` method on an existing crawler. This returns a Crawler instance with just the selected link(s). Calling `link()` gives you a special *Link*[24] object:

Listing 48-23
```
1 $linksCrawler = $crawler->selectLink('Go elsewhere...');
2 $link = $linksCrawler->link();
3
4 // or do this all at once
5 $link = $crawler->selectLink('Go elsewhere...')->link();
```

The *Link*[25] object has several useful methods to get more information about the selected link itself:

Listing 48-24
```
// return the proper URI that can be used to make another request
$uri = $link->getUri();
```

The `getUri()` is especially useful as it cleans the `href` value and transforms it into how it should really be processed. For example, for a link with `href="#foo"`, this would return the full URI of the current page suffixed with `#foo`. The return from `getUri()` is always a full URI that you can act on.

19. http://php.net/manual/en/class.domelement.php
20. http://php.net/manual/en/class.domelement.php
21. http://php.net/manual/en/class.domnode.php
22. http://php.net/manual/en/class.domdocument.php
23. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html#method_html
24. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Link.html
25. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Link.html

Forms

Special treatment is also given to forms. A `selectButton()` method is available on the Crawler which returns another Crawler that matches a button (`input[type=submit]`, `input[type=image]`, or a `button`) with the given text. This method is especially useful because you can use it to return a *Form*[26] object that represents the form that the button lives in:

Listing 48-25
```
1 $form = $crawler->selectButton('validate')->form();
2
3 // or "fill" the form fields with data
4 $form = $crawler->selectButton('validate')->form(array(
5     'name' => 'Ryan',
6 ));
```

The *Form*[27] object has lots of very useful methods for working with forms:

Listing 48-26
```
$uri = $form->getUri();

$method = $form->getMethod();
```

The *getUri()*[28] method does more than just return the `action` attribute of the form. If the form method is GET, then it mimics the browser's behavior and returns the `action` attribute followed by a query string of all of the form's values.

You can virtually set and get values on the form:

Listing 48-27
```
1  // set values on the form internally
2  $form->setValues(array(
3      'registration[username]' => 'symfonyfan',
4      'registration[terms]'    => 1,
5  ));
6
7  // get back an array of values - in the "flat" array like above
8  $values = $form->getValues();
9
10 // returns the values like PHP would see them,
11 // where "registration" is its own array
12 $values = $form->getPhpValues();
```

To work with multi-dimensional fields:

Listing 48-28
```
1 <form>
2     <input name="multi[]" />
3     <input name="multi[]" />
4     <input name="multi[dimensional]" />
5 </form>
```

Pass an array of values:

Listing 48-29
```
1 // Set a single field
2 $form->setValues(array('multi' => array('value')));
3
4 // Set multiple fields at once
```

26. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Form.html
27. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Form.html
28. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Form.html#method_getUri

```
5 $form->setValues(array('multi' => array(
6     1             => 'value',
7     'dimensional' => 'an other value'
8 )));
```

This is great, but it gets better! The `Form` object allows you to interact with your form like a browser, selecting radio values, ticking checkboxes, and uploading files:

Listing 48-30
```
1  $form['registration[username]']->setValue('symfonyfan');
2
3  // check or uncheck a checkbox
4  $form['registration[terms]']->tick();
5  $form['registration[terms]']->untick();
6
7  // select an option
8  $form['registration[birthday][year]']->select(1984);
9
10 // select many options from a "multiple" select
11 $form['registration[interests]']->select(array('symfony', 'cookies'));
12
13 // even fake a file upload
14 $form['registration[photo]']->upload('/path/to/lucas.jpg');
```

Using the Form Data

What's the point of doing all of this? If you're testing internally, you can grab the information off of your form as if it had just been submitted by using the PHP values:

Listing 48-31
```
$values = $form->getPhpValues();
$files = $form->getPhpFiles();
```

If you're using an external HTTP client, you can use the form to grab all of the information you need to create a POST request for the form:

Listing 48-32
```
1 $uri = $form->getUri();
2 $method = $form->getMethod();
3 $values = $form->getValues();
4 $files = $form->getFiles();
5
6 // now use some HTTP client and post using this information
```

One great example of an integrated system that uses all of this is *Goutte*[29]. Goutte understands the Symfony Crawler object and can use it to submit forms directly:

Listing 48-33
```
1 use Goutte\Client;
2
3 // make a real request to an external site
4 $client = new Client();
5 $crawler = $client->request('GET', 'https://github.com/login');
6
7 // select the form and fill in some values
8 $form = $crawler->selectButton('Log in')->form();
9 $form['login'] = 'symfonyfan';
```

29. https://github.com/FriendsOfPHP/Goutte

```
10 $form['password'] = 'anypass';
11
12 // submit that form
13 $crawler = $client->submit($form);
```

Selecting Invalid Choice Values

By default, choice fields (select, radio) have internal validation activated to prevent you from setting invalid values. If you want to be able to set invalid values, you can use the `disableValidation()` method on either the whole form or specific field(s):

Listing 48-34

```
1 // Disable validation for a specific field
2 $form['country']->disableValidation()->select('Invalid value');
3
4 // Disable validation for the whole form
5 $form->disableValidation();
6 $form['country']->select('Invalid value');
```

Chapter 49
The EventDispatcher Component

> The EventDispatcher component provides tools that allow your application components to communicate with each other by dispatching events and listening to them.

Introduction

Object-oriented code has gone a long way to ensuring code extensibility. By creating classes that have well defined responsibilities, your code becomes more flexible and a developer can extend them with subclasses to modify their behaviors. But if they want to share the changes with other developers who have also made their own subclasses, code inheritance is no longer the answer.

Consider the real-world example where you want to provide a plugin system for your project. A plugin should be able to add methods, or do something before or after a method is executed, without interfering with other plugins. This is not an easy problem to solve with single inheritance, and even if multiple inheritance was possible with PHP, it comes with its own drawbacks.

The Symfony EventDispatcher component implements the *Mediator*[1] pattern in a simple and effective way to make all these things possible and to make your projects truly extensible.

Take a simple example from *the HttpKernel component*. Once a `Response` object has been created, it may be useful to allow other elements in the system to modify it (e.g. add some cache headers) before it's actually used. To make this possible, the Symfony kernel throws an event - `kernel.response`. Here's how it works:

- A *listener* (PHP object) tells a central *dispatcher* object that it wants to listen to the `kernel.response` event;
- At some point, the Symfony kernel tells the *dispatcher* object to dispatch the `kernel.response` event, passing with it an `Event` object that has access to the `Response` object;
- The dispatcher notifies (i.e. calls a method on) all listeners of the `kernel.response` event, allowing each of them to make modifications to the `Response` object.

1. `https://en.wikipedia.org/wiki/Mediator_pattern`

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/event-dispatcher` on *Packagist*[2]);
- Use the official Git repository (*https://github.com/symfony/event-dispatcher*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

Events

When an event is dispatched, it's identified by a unique name (e.g. `kernel.response`), which any number of listeners might be listening to. An *`Event`*[3] instance is also created and passed to all of the listeners. As you'll see later, the `Event` object itself often contains data about the event being dispatched.

Naming Conventions

The unique event name can be any string, but optionally follows a few simple naming conventions:

- Use only lowercase letters, numbers, dots (`.`) and underscores (`_`);
- Prefix names with a namespace followed by a dot (e.g. `order.`, `user.*`);
- End names with a verb that indicates what action has been taken (e.g. `order.placed`).

Event Names and Event Objects

When the dispatcher notifies listeners, it passes an actual `Event` object to those listeners. The base `Event` class is very simple: it contains a method for stopping event propagation, but not much else.

> *Read "The Generic Event Object" for more information about this base event object.*

Often times, data about a specific event needs to be passed along with the `Event` object so that the listeners have the needed information. In such case, a special subclass that has additional methods for retrieving and overriding information can be passed when dispatching an event. For example, the `kernel.response` event uses a *`FilterResponseEvent`*[4], which contains methods to get and even replace the `Response` object.

The Dispatcher

The dispatcher is the central object of the event dispatcher system. In general, a single dispatcher is created, which maintains a registry of listeners. When an event is dispatched via the dispatcher, it notifies all listeners registered with that event:

Listing 49-1

```
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
```

2. https://packagist.org/packages/symfony/event-dispatcher
3. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html
4. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html

Connecting Listeners

To take advantage of an existing event, you need to connect a listener to the dispatcher so that it can be notified when the event is dispatched. A call to the dispatcher's `addListener()` method associates any valid PHP callable to an event:

Listing 49-2
```
$listener = new AcmeListener();
$dispatcher->addListener('acme.action', array($listener, 'onFooAction'));
```

The `addListener()` method takes up to three arguments:

1. The event name (string) that this listener wants to listen to;
2. A PHP callable that will be executed when the specified event is dispatched;
3. An optional priority integer (higher equals more important and therefore that the listener will be triggered earlier) that determines when a listener is triggered versus other listeners (defaults to `0`). If two listeners have the same priority, they are executed in the order that they were added to the dispatcher.

A *PHP callable*[5] is a PHP variable that can be used by the `call_user_func()` function and returns `true` when passed to the `is_callable()` function. It can be a `\Closure` instance, an object implementing an `__invoke` method (which is what closures are in fact), a string representing a function or an array representing an object method or a class method.

So far, you've seen how PHP objects can be registered as listeners. You can also register PHP *Closures*[6] as event listeners:

Listing 49-3
```
1  use Symfony\Component\EventDispatcher\Event;
2
3  $dispatcher->addListener('foo.action', function (Event $event) {
4      // will be executed when the foo.action event is dispatched
5  });
```

Once a listener is registered with the dispatcher, it waits until the event is notified. In the above example, when the `foo.action` event is dispatched, the dispatcher calls the `AcmeListener::onFooAction()` method and passes the `Event` object as the single argument:

Listing 49-4
```
 1  use Symfony\Component\EventDispatcher\Event;
 2
 3  class AcmeListener
 4  {
 5      // ...
 6
 7      public function onFooAction(Event $event)
 8      {
 9          // ... do something
10      }
11  }
```

The `$event` argument is the event class that was passed when dispatching the event. In many cases, a special event subclass is passed with extra information. You can check the documentation or implementation of each event to determine which instance is passed.

5. http://www.php.net/manual/en/language.pseudo-types.php#language.types.callback
6. http://php.net/manual/en/functions.anonymous.php

Registering Event Listeners in the Service Container

When you are using the *ContainerAwareEventDispatcher*[7] and the *DependencyInjection component*, you can use the *RegisterListenersPass*[8] to tag services as event listeners:

Listing 49-5

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\ParameterBag\ParameterBag;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\EventDispatcher\DependencyInjection\RegisterListenersPass;

$containerBuilder = new ContainerBuilder(new ParameterBag());
$containerBuilder->addCompilerPass(new RegisterListenersPass());

// register the event dispatcher service
$containerBuilder->setDefinition('event_dispatcher', new Definition(
    'Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher',
    array(new Reference('service_container'))
));

// register your event listener service
$listener = new Definition('AcmeListener');
$listener->addTag('kernel.event_listener', array(
    'event' => 'foo.action',
    'method' => 'onFooAction',
));
$containerBuilder->setDefinition('listener_service_id', $listener);

// register an event subscriber
$subscriber = new Definition('AcmeSubscriber');
$subscriber->addTag('kernel.event_subscriber');
$containerBuilder->setDefinition('subscriber_service_id', $subscriber);
```

By default, the listeners pass assumes that the event dispatcher's service id is `event_dispatcher`, that event listeners are tagged with the `kernel.event_listener` tag and that event subscribers are tagged with the `kernel.event_subscriber` tag. You can change these default values by passing custom values to the constructor of `RegisterListenersPass`.

Creating and Dispatching an Event

In addition to registering listeners with existing events, you can create and dispatch your own events. This is useful when creating third-party libraries and also when you want to keep different components of your own system flexible and decoupled.

Creating an Event Class

Suppose you want to create a new event - `order.placed` - that is dispatched each time a customer orders a product with your application. When dispatching this event, you'll pass a custom event instance that has access to the placed order. Start by creating this custom event class and documenting it:

Listing 49-6

```
namespace Acme\Store\Event;

use Symfony\Component\EventDispatcher\Event;
use Acme\Store\Order;
```

7. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
8. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/DependencyInjection/RegisterListenersPass.html

```
5
6 /**
7  * The order.placed event is dispatched each time an order is created
8  * in the system.
9  */
10 class OrderPlacedEvent extends Event
11 {
12     const NAME = 'order.placed';
13
14     protected $order;
15
16     public function __construct(Order $order)
17     {
18         $this->order = $order;
19     }
20
21     public function getOrder()
22     {
23         return $this->order;
24     }
25 }
```

Each listener now has access to the order via the `getOrder()` method.

If you don't need to pass any additional data to the event listeners, you can also use the default *Event*[9] class. In such case, you can document the event and its name in a generic `StoreEvents` class, similar to the *KernelEvents*[10] class.

Dispatch the Event

The *dispatch()*[11] method notifies all listeners of the given event. It takes two arguments: the name of the event to dispatch and the `Event` instance to pass to each listener of that event:

Listing 49-7

```
1 use Acme\Store\Order;
2 use Acme\Store\Event\OrderPlacedEvent;
3
4 // the order is somehow created or retrieved
5 $order = new Order();
6 // ...
7
8 // create the OrderPlacedEvent and dispatch it
9 $event = new OrderPlacedEvent($order);
10 $dispatcher->dispatch(OrderPlacedEvent::NAME, $event);
```

Notice that the special `OrderPlacedEvent` object is created and passed to the `dispatch()` method. Now, any listener to the `order.placed` event will receive the `OrderPlacedEvent`.

Using Event Subscribers

The most common way to listen to an event is to register an *event listener* with the dispatcher. This listener can listen to one or more events and is notified each time those events are dispatched.

9. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html
10. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelEvents.html
11. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventDispatcher.html#method_dispatch

Another way to listen to events is via an *event subscriber*. An event subscriber is a PHP class that's able to tell the dispatcher exactly which events it should subscribe to. It implements the *EventSubscriberInterface*[12] interface, which requires a single static method called *getSubscribedEvents()*[13]. Take the following example of a subscriber that subscribes to the `kernel.response` and `order.placed` events:

Listing 49-8

```
namespace Acme\Store\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
use Symfony\Component\HttpKernel\KernelEvents;
use Acme\Store\Event\OrderPlacedEvent;

class StoreSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            KernelEvents::RESPONSE => array(
                array('onKernelResponsePre', 10),
                array('onKernelResponsePost', -10),
            ),
            OrderPlacedEvent::NAME => 'onStoreOrder',
        );
    }

    public function onKernelResponsePre(FilterResponseEvent $event)
    {
        // ...
    }

    public function onKernelResponsePost(FilterResponseEvent $event)
    {
        // ...
    }

    public function onStoreOrder(OrderPlacedEvent $event)
    {
        // ...
    }
}
```

This is very similar to a listener class, except that the class itself can tell the dispatcher which events it should listen to. To register a subscriber with the dispatcher, use the *addSubscriber()*[14] method:

Listing 49-9

```
use Acme\Store\Event\StoreSubscriber;
// ...

$subscriber = new StoreSubscriber();
$dispatcher->addSubscriber($subscriber);
```

The dispatcher will automatically register the subscriber for each event returned by the `getSubscribedEvents()` method. This method returns an array indexed by event names and whose values are either the method name to call or an array composed of the method name to call and a priority. The example above shows how to register several listener methods for the same event in subscriber

12. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventSubscriberInterface.html
13. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventSubscriberInterface.html#method_getSubscribedEvents
14. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventDispatcher.html#method_addSubscriber

and also shows how to pass the priority of each listener method. The higher the priority, the earlier the method is called. In the above example, when the `kernel.response` event is triggered, the methods `onKernelResponsePre()` and `onKernelResponsePost()` are called in that order.

Stopping Event Flow/Propagation

In some cases, it may make sense for a listener to prevent any other listeners from being called. In other words, the listener needs to be able to tell the dispatcher to stop all propagation of the event to future listeners (i.e. to not notify any more listeners). This can be accomplished from inside a listener via the *stopPropagation()*[15] method:

Listing 49-10
```
use Acme\Store\Event\OrderPlacedEvent;

public function onStoreOrder(OrderPlacedEvent $event)
{
    // ...

    $event->stopPropagation();
}
```

Now, any listeners to `order.placed` that have not yet been called will *not* be called.

It is possible to detect if an event was stopped by using the *isPropagationStopped()*[16] method which returns a boolean value:

Listing 49-11
```
// ...
$dispatcher->dispatch('foo.event', $event);
if ($event->isPropagationStopped()) {
    // ...
}
```

EventDispatcher Aware Events and Listeners

The `EventDispatcher` always passes the dispatched event, the event's name and a reference to itself to the listeners. This can be used in some advanced usages of the `EventDispatcher` like dispatching other events in listeners, event chaining or even lazy loading of more listeners into the dispatcher object as shown in the following examples.

This can lead to some advanced applications of the `EventDispatcher` including dispatching other events inside listeners, chaining events or even lazy loading listeners into the dispatcher object.

Dispatcher Shortcuts

If you do not need a custom event object, you can simply rely on a plain *Event*[17] object. You do not even need to pass this to the dispatcher as it will create one by default unless you specifically pass one:

Listing 49-12
```
$dispatcher->dispatch('order.placed');
```

Moreover, the event dispatcher always returns whichever event object that was dispatched, i.e. either the event that was passed or the event that was created internally by the dispatcher. This allows for nice shortcuts:

Listing 49-13

15. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html#method_stopPropagation
16. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html#method_isPropagationStopped
17. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html

```
if (!$dispatcher->dispatch('foo.event')->isPropagationStopped()) {
    // ...
}
```

Or:

Listing 49-14
```
$event = new OrderPlacedEvent($order);
$order = $dispatcher->dispatch('bar.event', $event)->getOrder();
```

and so on.

Event Name Introspection

The `EventDispatcher` instance, as well as the name of the event that is dispatched, are passed as arguments to the listener:

Listing 49-15
```
1  use Symfony\Component\EventDispatcher\Event;
2  use Symfony\Component\EventDispatcher\EventDispatcherInterface;
3
4  class Foo
5  {
6      public function myEventListener(Event $event, $eventName, EventDispatcherInterface
7  $dispatcher)
8      {
9          // ... do something with the event name
10     }
   }
```

Other Dispatchers

Besides the commonly used `EventDispatcher`, the component comes with some other dispatchers:

- *The Container Aware Event Dispatcher*
- *The Immutable Event Dispatcher*
- *The Traceable Event Dispatcher* (provided by the *HttpKernel component*)

Chapter 50
The Container Aware Event Dispatcher

Introduction

The *ContainerAwareEventDispatcher*[1] is a special `EventDispatcher` implementation which is coupled to the service container that is part of *the DependencyInjection component*. It allows services to be specified as event listeners making the `EventDispatcher` extremely powerful.

Services are lazy loaded meaning the services attached as listeners will only be created if an event is dispatched that requires those listeners.

Setup

Setup is straightforward by injecting a *ContainerInterface*[2] into the *ContainerAwareEventDispatcher*[3]:

Listing 50-1

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher;

$container = new ContainerBuilder();
$dispatcher = new ContainerAwareEventDispatcher($container);
```

Adding Listeners

The `ContainerAwareEventDispatcher` can either load specified services directly or services that implement *EventSubscriberInterface*[4].

1. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerInterface.html
3. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
4. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

The following examples assume the service container has been loaded with any services that are mentioned.

Services must be marked as public in the container.

Adding Services

To connect existing service definitions, use the *addListenerService()*[5] method where the `$callback` is an array of `array($serviceId, $methodName)`:

Listing 50-2
```
$dispatcher->addListenerService($eventName, array('foo', 'logListener'));
```

Adding Subscriber Services

Event subscribers can be added using the *addSubscriberService()*[6] method where the first argument is the service ID of the subscriber service, and the second argument is the service's class name (which must implement *EventSubscriberInterface*[7]) as follows:

Listing 50-3
```
$dispatcher->addSubscriberService(
    'kernel.store_subscriber',
    'StoreSubscriber'
);
```

The `EventSubscriberInterface` is exactly as you would expect:

Listing 50-4
```
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
// ...

class StoreSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            'kernel.response' => array(
                array('onKernelResponsePre', 10),
                array('onKernelResponsePost', 0),
            ),
            'store.order'     => array('onStoreOrder', 0),
        );
    }

    public function onKernelResponsePre(FilterResponseEvent $event)
    {
        // ...
    }

    public function onKernelResponsePost(FilterResponseEvent $event)
    {
        // ...
    }

```

5. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#method_addListenerService
6. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#method_addSubscriberService
7. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

```
27      public function onStoreOrder(FilterOrderEvent $event)
28      {
29          // ...
30      }
31  }
```

Chapter 51
The Generic Event Object

The base *Event*[1] class provided by the EventDispatcher component is deliberately sparse to allow the creation of API specific event objects by inheritance using OOP. This allows for elegant and readable code in complex applications.

The *GenericEvent*[2] is available for convenience for those who wish to use just one event object throughout their application. It is suitable for most purposes straight out of the box, because it follows the standard observer pattern where the event object encapsulates an event 'subject', but has the addition of optional extra arguments.

GenericEvent[3] has a simple API in addition to the base class *Event*[4]

- *__construct()*[5]: Constructor takes the event subject and any arguments;
- *getSubject()*[6]: Get the subject;
- *setArgument()*[7]: Sets an argument by key;
- *setArguments()*[8]: Sets arguments array;
- *getArgument()*[9]: Gets an argument by key;
- *getArguments()*[10]: Getter for all arguments;
- *hasArgument()*[11]: Returns true if the argument key exists;

The `GenericEvent` also implements *ArrayAccess*[12] on the event arguments which makes it very convenient to pass extra arguments regarding the event subject.

The following examples show use-cases to give a general idea of the flexibility. The examples assume event listeners have been added to the dispatcher.

Simply passing a subject:

1. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html
2. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html
3. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html
4. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method___construct
6. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_getSubject
7. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_setArgument
8. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_setArguments
9. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_getArgument
10. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_getArguments
11. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/GenericEvent.html#method_hasArgument
12. http://php.net/manual/en/class.arrayaccess.php

Listing 51-1

```
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject);
$dispatcher->dispatch('foo', $event);

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if ($event->getSubject() instanceof Foo) {
            // ...
        }
    }
}
```

Passing and processing arguments using the *ArrayAccess*[13] API to access the event arguments:

Listing 51-2

```
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent(
    $subject,
    array('type' => 'foo', 'counter' => 0)
);
$dispatcher->dispatch('foo', $event);

var_dump($event['counter']);

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if (isset($event['type']) && $event['type'] === 'foo') {
            // ... do something
        }

        $event['counter']++;
    }
}
```

Filtering data:

Listing 51-3

```
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject, array('data' => 'Foo'));
$dispatcher->dispatch('foo', $event);

var_dump($event['data']);

class FooListener
{
    public function filter(GenericEvent $event)
    {
        $event['data'] = strtolower($event['data']);
    }
}
```

13. http://php.net/manual/en/class.arrayaccess.php

Chapter 52

The Immutable Event Dispatcher

The *ImmutableEventDispatcher*[1] is a locked or frozen event dispatcher. The dispatcher cannot register new listeners or subscribers.

The `ImmutableEventDispatcher` takes another event dispatcher with all the listeners and subscribers. The immutable dispatcher is just a proxy of this original dispatcher.

To use it, first create a normal dispatcher (`EventDispatcher` or `ContainerAwareEventDispatcher`) and register some listeners or subscribers:

Listing 52-1

```
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
$dispatcher->addListener('foo.action', function ($event) {
    // ...
});

// ...
```

Now, inject that into an `ImmutableEventDispatcher`:

Listing 52-2

```
use Symfony\Component\EventDispatcher\ImmutableEventDispatcher;
// ...

$immutableDispatcher = new ImmutableEventDispatcher($dispatcher);
```

You'll need to use this new dispatcher in your project.

If you are trying to execute one of the methods which modifies the dispatcher (e.g. `addListener`), a `BadMethodCallException` is thrown.

1. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/ImmutableEventDispatcher.html

Chapter 53
The Traceable Event Dispatcher

The *TraceableEventDispatcher*[1] is an event dispatcher that wraps any other event dispatcher and can then be used to determine which event listeners have been called by the dispatcher. Pass the event dispatcher to be wrapped and an instance of the *Stopwatch*[2] to its constructor:

Listing 53-1
```
use Symfony\Component\EventDispatcher\Debug\TraceableEventDispatcher;
use Symfony\Component\Stopwatch\Stopwatch;

// the event dispatcher to debug
$eventDispatcher = ...;

$traceableEventDispatcher = new TraceableEventDispatcher(
    $eventDispatcher,
    new Stopwatch()
);
```

Now, the `TraceableEventDispatcher` can be used like any other event dispatcher to register event listeners and dispatch events:

Listing 53-2
```
// ...

// register an event listener
$eventListener = ...;
$priority = ...;
$traceableEventDispatcher->addListener(
    'event.the_name',
    $eventListener,
    $priority
);

// dispatch an event
$event = ...;
$traceableEventDispatcher->dispatch('event.the_name', $event);
```

1. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Debug/TraceableEventDispatcher.html
2. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html

After your application has been processed, you can use the *getCalledListeners()*[3] method to retrieve an array of event listeners that have been called in your application. Similarly, the *getNotCalledListeners()*[4] method returns an array of event listeners that have not been called:

Listing 53-3
```
// ...

$calledListeners = $traceableEventDispatcher->getCalledListeners();
$notCalledListeners = $traceableEventDispatcher->getNotCalledListeners();
```

3. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Debug/TraceableEventDispatcherInterface.html#method_getCalledListeners
4. http://api.symfony.com/3.0/Symfony/Component/EventDispatcher/Debug/TraceableEventDispatcherInterface.html#method_getNotCalledListeners

Chapter 54
The ExpressionLanguage Component

> The ExpressionLanguage component provides an engine that can compile and evaluate expressions. An expression is a one-liner that returns a value (mostly, but not limited to, Booleans).

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/expression-language` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/expression-language*).

How can the Expression Engine Help Me?

The purpose of the component is to allow users to use expressions inside configuration for more complex logic. For some examples, the Symfony2 Framework uses expressions in security, for validation rules and in route matching.

Besides using the component in the framework itself, the ExpressionLanguage component is a perfect candidate for the foundation of a *business rule engine*. The idea is to let the webmaster of a website configure things in a dynamic way without using PHP and without introducing security problems:

Listing 54-1

```
# Get the special price if
user.getGroup() in ['good_customers', 'collaborator']

# Promote article to the homepage when
article.commentCount > 100 and article.category not in ["misc"]

# Send an alert when
product.stock < 15
```

1. https://packagist.org/packages/symfony/expression-language

Expressions can be seen as a very restricted PHP sandbox and are immune to external injections as you must explicitly declare which variables are available in an expression.

Usage

The ExpressionLanguage component can compile and evaluate expressions. Expressions are one-liners that often return a Boolean, which can be used by the code executing the expression in an `if` statement. A simple example of an expression is `1 + 2`. You can also use more complicated expressions, such as `someArray[3].someMethod('bar')`.

The component provides 2 ways to work with expressions:

- **evaluation**: the expression is evaluated without being compiled to PHP;
- **compile**: the expression is compiled to PHP, so it can be cached and evaluated.

The main class of the component is *ExpressionLanguage*[2]:

Listing 54-2

```
1 use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
2
3 $language = new ExpressionLanguage();
4
5 var_dump($language->evaluate('1 + 2')); // displays 3
6
7 var_dump($language->compile('1 + 2')); // displays (1 + 2)
```

Expression Syntax

See *The Expression Syntax* to learn the syntax of the ExpressionLanguage component.

Passing in Variables

You can also pass variables into the expression, which can be of any valid PHP type (including objects):

Listing 54-3

```
1  use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
2
3  $language = new ExpressionLanguage();
4
5  class Apple
6  {
7      public $variety;
8  }
9
10 $apple = new Apple();
11 $apple->variety = 'Honeycrisp';
12
13 var_dump($language->evaluate(
14     'fruit.variety',
15     array(
16         'fruit' => $apple,
17     )
18 ));
```

2. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html

This will print "Honeycrisp". For more information, see the *The Expression Syntax* entry, especially Working with Objects and Working with Arrays.

Caching

The component provides some different caching strategies, read more about them in *Caching Expressions Using Parser Caches*.

Chapter 55

The Expression Syntax

The ExpressionLanguage component uses a specific syntax which is based on the expression syntax of Twig. In this document, you can find all supported syntaxes.

Supported Literals

The component supports:

- **strings** - single and double quotes (e.g. `'hello'`)
- **numbers** - e.g. `103`
- **arrays** - using JSON-like notation (e.g. `[1, 2]`)
- **hashes** - using JSON-like notation (e.g. `{ foo: 'bar' }`)
- **booleans** - `true` and `false`
- **null** - `null`

A backslash (`\`) must be escaped by 4 backslashes (`\\\\`) in a string and 8 backslashes (`\\\\\\\\`) in a regex:

Listing 55-1
```
echo $language->evaluate('"\\\\"'); // prints \
$language->evaluate('"a\\\\b" matches "/^a\\\\\\\\b$/"'); // returns true
```

Control characters (e.g. `\n`) in expressions are replaced with whitespace. To avoid this, escape the sequence with a single backslash (e.g. `\\n`).

Working with Objects

When passing objects into an expression, you can use different syntaxes to access properties and call methods on the object.

Accessing Public Properties

Public properties on objects can be accessed by using the `.` syntax, similar to JavaScript:

Listing 55-2

```
class Apple
{
    public $variety;
}

$apple = new Apple();
$apple->variety = 'Honeycrisp';

var_dump($language->evaluate(
    'fruit.variety',
    array(
        'fruit' => $apple,
    )
));
```

This will print out `Honeycrisp`.

Calling Methods

The `.` syntax can also be used to call methods on an object, similar to JavaScript:

Listing 55-3

```
class Robot
{
    public function sayHi($times)
    {
        $greetings = array();
        for ($i = 0; $i < $times; $i++) {
            $greetings[] = 'Hi';
        }

        return implode(' ', $greetings).'!';
    }
}

$robot = new Robot();

var_dump($language->evaluate(
    'robot.sayHi(3)',
    array(
        'robot' => $robot,
    )
));
```

This will print out `Hi Hi Hi!`.

Working with Functions

You can also use registered functions in the expression by using the same syntax as PHP and JavaScript. The ExpressionLanguage component comes with one function by default: `constant()`, which will return the value of the PHP constant:

Listing 55-4

```
1 define('DB_USER', 'root');
2
3 var_dump($language->evaluate(
4     'constant("DB_USER")'
5 ));
```

This will print out `root`.

To read how to register your own functions to use in an expression, see "*Extending the ExpressionLanguage*".

Working with Arrays

If you pass an array into an expression, use the `[]` syntax to access array keys, similar to JavaScript:

Listing 55-5

```
1 $data = array('life' => 10, 'universe' => 10, 'everything' => 22);
2
3 var_dump($language->evaluate(
4     'data["life"] + data["universe"] + data["everything"]',
5     array(
6         'data' => $data,
7     )
8 ));
```

This will print out `42`.

Supported Operators

The component comes with a lot of operators:

Arithmetic Operators

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulus)
- `**` (pow)

For example:

Listing 55-6

```
1 var_dump($language->evaluate(
2     'life + universe + everything',
3     array(
4         'life' => 10,
5         'universe' => 10,
6         'everything' => 22,
7     )
8 ));
```

This will print out `42`.

Bitwise Operators

- `&` (and)
- `|` (or)
- `^` (xor)

Comparison Operators

- `==` (equal)
- `===` (identical)
- `!=` (not equal)
- `!==` (not identical)
- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `matches` (regex match)

To test if a string does *not* match a regex, use the logical `not` operator in combination with the `matches` operator:

Listing 55-7
```
$language->evaluate('not ("foo" matches "/bar/")'); // returns true
```

You must use parenthesis because the unary operator `not` has precedence over the binary operator `matches`.

Examples:

Listing 55-8
```
$ret1 = $language->evaluate(
    'life == everything',
    array(
        'life' => 10,
        'universe' => 10,
        'everything' => 22,
    )
);

$ret2 = $language->evaluate(
    'life > everything',
    array(
        'life' => 10,
        'universe' => 10,
        'everything' => 22,
    )
);
```

Both variables would be set to `false`.

Logical Operators

- `not` or `!`
- `and` or `&&`
- `or` or `||`

For example:

Listing 55-9
```
$ret = $language->evaluate(
    'life < universe or life < everything',
    array(
        'life' => 10,
        'universe' => 10,
        'everything' => 22,
    )
);
```

This `$ret` variable will be set to `true`.

String Operators

- `~` (concatenation)

For example:

Listing 55-10
```
var_dump($language->evaluate(
    'firstName~" "~lastName',
    array(
        'firstName' => 'Arthur',
        'lastName' => 'Dent',
    )
));
```

This would print out `Arthur Dent`.

Array Operators

- `in` (contain)
- `not in` (does not contain)

For example:

Listing 55-11
```
class User
{
    public $group;
}

$user = new User();
$user->group = 'human_resources';

$inGroup = $language->evaluate(
    'user.group in ["human_resources", "marketing"]',
    array(
        'user' => $user,
    )
);
```

The `$inGroup` would evaluate to `true`.

Numeric Operators

- `..` (range)

For example:

Listing 55-12
```
class User
{
    public $age;
}

$user = new User();
$user->age = 34;

$language->evaluate(
    'user.age in 18..45',
    array(
        'user' => $user,
    )
);
```

This will evaluate to `true`, because `user.age` is in the range from `18` to `45`.

Ternary Operators

- `foo ? 'yes' : 'no'`
- `foo ?: 'no'` (equal to `foo ? foo : 'no'`)
- `foo ? 'yes'` (equal to `foo ? 'yes' : ''`)

Chapter 56

Extending the ExpressionLanguage

The ExpressionLanguage can be extended by adding custom functions. For instance, in the Symfony Framework, the security has custom functions to check the user's role.

If you want to learn how to use functions in an expression, read "Working with Functions".

Registering Functions

Functions are registered on each specific `ExpressionLanguage` instance. That means the functions can be used in any expression executed by that instance.

To register a function, use *`register()`*[1]. This method has 3 arguments:

- **name** - The name of the function in an expression;
- **compiler** - A function executed when compiling an expression using the function;
- **evaluator** - A function executed when the expression is evaluated.

Listing 56-1

```
use Symfony\Component\ExpressionLanguage\ExpressionLanguage;

$language = new ExpressionLanguage();
$language->register('lowercase', function ($str) {
    return sprintf('(is_string(%1$s) ? strtolower(%1$s) : %1$s)', $str);
}, function ($arguments, $str) {
    if (!is_string($str)) {
        return $str;
    }

    return strtolower($str);
});
```

1. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html#method_register

```
13
14 var_dump($language->evaluate('lowercase("HELLO")'));
```

This will print `hello`. Both the **compiler** and **evaluator** are passed an `arguments` variable as their first argument, which is equal to the second argument to `evaluate()` or `compile()` (e.g. the "values" when evaluating or the "names" if compiling).

Using Expression Providers

When you use the `ExpressionLanguage` class in your library, you often want to add custom functions. To do so, you can create a new expression provider by creating a class that implements *`ExpressionFunctionProviderInterface`*[2].

This interface requires one method: *`getFunctions()`*[3], which returns an array of expression functions (instances of *`ExpressionFunction`*[4]) to register.

Listing 56-2

```
 1 use Symfony\Component\ExpressionLanguage\ExpressionFunction;
 2 use Symfony\Component\ExpressionLanguage\ExpressionFunctionProviderInterface;
 3
 4 class StringExpressionLanguageProvider implements ExpressionFunctionProviderInterface
 5 {
 6     public function getFunctions()
 7     {
 8         return array(
 9             new ExpressionFunction('lowercase', function ($str) {
10                 return sprintf('(is_string(%1$s) ? strtolower(%1$s) : %1$s)', $str);
11             }, function ($arguments, $str) {
12                 if (!is_string($str)) {
13                     return $str;
14                 }
15
16                 return strtolower($str);
17             }),
18         );
19     }
20 }
```

You can register providers using *`registerProvider()`*[5] or by using the second argument of the constructor:

Listing 56-3

```
 1 use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
 2
 3 // using the constructor
 4 $language = new ExpressionLanguage(null, array(
 5     new StringExpressionLanguageProvider(),
 6     // ...
 7 ));
 8
 9 // using registerProvider()
10 $language->registerProvider(new StringExpressionLanguageProvider());
```

2. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionFunctionProviderInterface.html
3. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionFunctionProviderInterface.html#method_getFunctions
4. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionFunction.html
5. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html#method_registerProvider

It is recommended to create your own `ExpressionLanguage` class in your library. Now you can add the extension by overriding the constructor:

Listing 56-4

```
use Symfony\Component\ExpressionLanguage\ExpressionLanguage as
BaseExpressionLanguage;
use Symfony\Component\ExpressionLanguage\ParserCache\ParserCacheInterface;

class ExpressionLanguage extends BaseExpressionLanguage
{
    public function __construct(ParserCacheInterface $parser = null, array
$providers = array())
    {
        // prepend the default provider to let users override it easily
        array_unshift($providers, new StringExpressionLanguageProvider());

        parent::__construct($parser, $providers);
    }
}
```

Chapter 57

Caching Expressions Using Parser Caches

The ExpressionLanguage component already provides a *compile()*[1] method to be able to cache the expressions in plain PHP. But internally, the component also caches the parsed expressions, so duplicated expressions can be compiled/evaluated quicker.

The Workflow

Both *evaluate()*[2] and `compile()` need to do some things before each can provide the return values. For `evaluate()`, this overhead is even bigger.

Both methods need to tokenize and parse the expression. This is done by the *parse()*[3] method. It returns a *ParsedExpression*[4]. Now, the `compile()` method just returns the string conversion of this object. The `evaluate()` method needs to loop through the "nodes" (pieces of an expression saved in the `ParsedExpression`) and evaluate them on the fly.

To save time, the `ExpressionLanguage` caches the `ParsedExpression` so it can skip the tokenize and parse steps with duplicate expressions. The caching is done by a *ParserCacheInterface*[5] instance (by default, it uses an *ArrayParserCache*[6]). You can customize this by creating a custom `ParserCache` and injecting this in the object using the constructor:

Listing 57-1

```
1 use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
2 use Acme\ExpressionLanguage\ParserCache\MyDatabaseParserCache;
3
4 $cache = new MyDatabaseParserCache(...);
5 $language = new ExpressionLanguage($cache);
```

1. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html#method_compile
2. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html#method_evaluate
3. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ExpressionLanguage.html#method_parse
4. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ParsedExpression.html
5. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ParserCache/ParserCacheInterface.html
6. http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/ParserCache/ArrayParserCache.html

The *DoctrineBridge*[7] provides a Parser Cache implementation using the *doctrine cache library*[8], which gives you caching for all sorts of cache strategies, like Apc, Filesystem and Memcached.

Using Parsed and Serialized Expressions

Both `evaluate()` and `compile()` can handle `ParsedExpression` and `SerializedParsedExpression`:

Listing 57-2
```
// ...

// the parse() method returns a ParsedExpression
$expression = $language->parse('1 + 4', array());

var_dump($language->evaluate($expression)); // prints 5
```

Listing 57-3
```
use Symfony\Component\ExpressionLanguage\SerializedParsedExpression;
// ...

$expression = new SerializedParsedExpression(
    '1 + 4',
    serialize($language->parse('1 + 4', array())->getNodes())
);

var_dump($language->evaluate($expression)); // prints 5
```

7. `https://github.com/symfony/doctrine-bridge`
8. `http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/caching.html`

Chapter 58

The Filesystem Component

> The Filesystem component provides basic utilities for the filesystem.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/filesystem` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/filesystem*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The *`Filesystem`*[2] class is the unique endpoint for filesystem operations:

Listing 58-1

```
1  use Symfony\Component\Filesystem\Filesystem;
2  use Symfony\Component\Filesystem\Exception\IOExceptionInterface;
3
4  $fs = new Filesystem();
5
6  try {
7      $fs->mkdir('/tmp/random/dir/'.mt_rand());
8  } catch (IOExceptionInterface $e) {
9      echo "An error occurred while creating your directory at ".$e->getPath();
10 }
```

1. https://packagist.org/packages/symfony/filesystem
2. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html

Methods *mkdir()*[3], *exists()*[4], *touch()*[5], *remove()*[6], *chmod()*[7], *chown()*[8] and *chgrp()*[9] can receive a string, an array or any object implementing *Traversable*[10] as the target argument.

mkdir

mkdir()[11] creates a directory. On POSIX filesystems, directories are created with a default mode value *0777*. You can use the second argument to set your own mode:

Listing 58-2
```
$fs->mkdir('/tmp/photos', 0700);
```

You can pass an array or any *Traversable*[12] object as the first argument.

exists

exists()[13] checks for the presence of all files or directories and returns `false` if a file is missing:

Listing 58-3
```
1  // this directory exists, return true
2  $fs->exists('/tmp/photos');
3
4  // rabbit.jpg exists, bottle.png does not exists, return false
5  $fs->exists(array('rabbit.jpg', 'bottle.png'));
```

You can pass an array or any *Traversable*[14] object as the first argument.

copy

copy()[15] is used to copy files. If the target already exists, the file is copied only if the source modification date is later than the target. This behavior can be overridden by the third boolean argument:

Listing 58-4
```
1  // works only if image-ICC has been modified after image.jpg
2  $fs->copy('image-ICC.jpg', 'image.jpg');
3
```

3. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_mkdir
4. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_exists
5. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_touch
6. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_remove
7. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chmod
8. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chown
9. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chgrp
10. http://php.net/manual/en/class.traversable.php
11. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_mkdir
12. http://php.net/manual/en/class.traversable.php
13. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_exists
14. http://php.net/manual/en/class.traversable.php
15. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_copy

```
4 // image.jpg will be overridden
5 $fs->copy('image-ICC.jpg', 'image.jpg', true);
```

touch

touch()[16] sets access and modification time for a file. The current time is used by default. You can set your own with the second argument. The third argument is the access time:

Listing 58-5
```
1 // set modification time to the current timestamp
2 $fs->touch('file.txt');
3 // set modification time 10 seconds in the future
4 $fs->touch('file.txt', time() + 10);
5 // set access time 10 seconds in the past
6 $fs->touch('file.txt', time(), time() - 10);
```

You can pass an array or any *Traversable*[17] object as the first argument.

chown

chown()[18] is used to change the owner of a file. The third argument is a boolean recursive option:

Listing 58-6
```
// set the owner of the lolcat video to www-data
$fs->chown('lolcat.mp4', 'www-data');
// change the owner of the video directory recursively
$fs->chown('/video', 'www-data', true);
```

You can pass an array or any *Traversable*[19] object as the first argument.

chgrp

chgrp()[20] is used to change the group of a file. The third argument is a boolean recursive option:

Listing 58-7
```
// set the group of the lolcat video to nginx
$fs->chgrp('lolcat.mp4', 'nginx');
// change the group of the video directory recursively
$fs->chgrp('/video', 'nginx', true);
```

You can pass an array or any *Traversable*[21] object as the first argument.

16. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_touch
17. http://php.net/manual/en/class.traversable.php
18. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chown
19. http://php.net/manual/en/class.traversable.php
20. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chgrp
21. http://php.net/manual/en/class.traversable.php

chmod

chmod()[22] is used to change the mode of a file. The fourth argument is a boolean recursive option:

Listing 58-8
```
// set the mode of the video to 0600
$fs->chmod('video.ogg', 0600);
// change the mod of the src directory recursively
$fs->chmod('src', 0700, 0000, true);
```

You can pass an array or any *Traversable*[23] object as the first argument.

remove

remove()[24] is used to remove files, symlinks, directories easily:

Listing 58-9
```
$fs->remove(array('symlink', '/path/to/directory', 'activity.log'));
```

You can pass an array or any *Traversable*[25] object as the first argument.

rename

rename()[26] is used to rename files and directories:

Listing 58-10
```
// rename a file
$fs->rename('/tmp/processed_video.ogg', '/path/to/store/video_647.ogg');
// rename a directory
$fs->rename('/tmp/files', '/path/to/store/files');
```

symlink

symlink()[27] creates a symbolic link from the target to the destination. If the filesystem does not support symbolic links, a third boolean argument is available:

Listing 58-11
```
1 // create a symbolic link
2 $fs->symlink('/path/to/source', '/path/to/destination');
3 // duplicate the source directory if the filesystem
4 // does not support symbolic links
5 $fs->symlink('/path/to/source', '/path/to/destination', true);
```

makePathRelative

makePathRelative()[28] returns the relative path of a directory given another one:

22. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_chmod
23. http://php.net/manual/en/class.traversable.php
24. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_remove
25. http://php.net/manual/en/class.traversable.php
26. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_rename
27. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_symlink
28. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_makePathRelative

Listing 58-12
```
// returns '../'
$fs->makePathRelative(
    '/var/lib/symfony/src/Symfony/',
    '/var/lib/symfony/src/Symfony/Component'
);
// returns 'videos/'
$fs->makePathRelative('/tmp/videos', '/tmp')
```

mirror

mirror()[29] mirrors a directory:

Listing 58-13
```
$fs->mirror('/path/to/source', '/path/to/target');
```

isAbsolutePath

isAbsolutePath()[30] returns `true` if the given path is absolute, `false` otherwise:

Listing 58-14
```
// return true
$fs->isAbsolutePath('/tmp');
// return true
$fs->isAbsolutePath('c:\\Windows');
// return false
$fs->isAbsolutePath('tmp');
// return false
$fs->isAbsolutePath('../dir');
```

dumpFile

dumpFile()[31] allows you to dump contents to a file. It does this in an atomic manner: it writes a temporary file first and then moves it to the new file location when it's finished. This means that the user will always see either the complete old file or complete new file (but never a partially-written file):

Listing 58-15
```
$fs->dumpFile('file.txt', 'Hello World');
```

The `file.txt` file contains `Hello World` now.

A desired file mode can be passed as the third argument.

Error Handling

Whenever something wrong happens, an exception implementing *ExceptionInterface*[32] or *IOExceptionInterface*[33] is thrown.

An *IOException*[34] is thrown if directory creation fails.

29. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_mirror
30. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_isAbsolutePath
31. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Filesystem.html#method_dumpFile
32. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Exception/ExceptionInterface.html
33. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Exception/IOExceptionInterface.html

34. http://api.symfony.com/3.0/Symfony/Component/Filesystem/Exception/IOException.html

Chapter 59

LockHandler

What is a Lock?

File locking is a mechanism that restricts access to a computer file by allowing only one user or process access at any specific time. This mechanism was introduced a few decades ago for mainframes, but continues being useful for modern applications.

Symfony provides a LockHelper to help you use locks in your project.

Usage

The lock handler only works if you're using just one server. If you have several hosts, you must not use this helper.

A lock can be used, for example, to allow only one instance of a command to run.

Listing 59-1

```
use Symfony\Component\Filesystem\LockHandler;

$lockHandler = new LockHandler('hello.lock');
if (!$lockHandler->lock()) {
    // the resource "hello" is already locked by another process

    return 0;
}
```

The first argument of the constructor is a string that it will use as part of the name of the file used to create the lock on the local filesystem. A best practice for Symfony commands is to use the command name, such as `acme:my-command`. `LockHandler` sanitizes the contents of the string before creating the file, so you can pass any value for this argument.

The `.lock` extension is optional, but it's a common practice to include it. This will make it easier to find lock files on the filesystem. Moreover, to avoid name collisions, `LockHandler` also appends a hash to the name of the lock file.

By default, the lock will be created in the temporary directory, but you can optionally select the directory where locks are created by passing it as the second argument of the constructor.

The *`lock()`*[1] method tries to acquire the lock. If the lock is acquired, the method returns `true`, `false` otherwise. If the `lock` method is called several times on the same instance it will always return `true` if the lock was acquired on the first call.

You can pass an optional blocking argument as the first argument to the `lock()` method, which defaults to `false`. If this is set to `true`, your PHP code will wait indefinitely until the lock is released by another process.

Be aware of the fact that the resource lock is automatically released as soon as PHP applies the garbage-collection process to the `LockHandler` object. This means that if you refactor the first example shown in this article as follows:

Listing 59-2
```
use Symfony\Component\Filesystem\LockHandler;

 if (!(new LockHandler('hello.lock'))->lock()) {
    // the resource "hello" is already locked by another process

    return 0;
}
```

Now the code won't work as expected because PHP's garbage collection mechanism removes the reference to the `LockHandler` object and thus, the lock is released just after it's been created.

Another alternative way to release the lock explicitly when needed is to use the *`release()`*[2] method.

1. http://api.symfony.com/3.0/Symfony/Component/Filesystem/LockHandler.html#method_lock
2. http://api.symfony.com/3.0/Symfony/Component/Filesystem/LockHandler.html#method_release

Chapter 60
The Finder Component

> The Finder component finds files and directories via an intuitive fluent interface.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/finder` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/finder*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The *`Finder`*[2] class finds files and/or directories:

Listing 60-1

```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    // Dump the absolute path
    var_dump($file->getRealpath());

    // Dump the relative path to the file, omitting the filename
    var_dump($file->getRelativePath());
```

1. https://packagist.org/packages/symfony/finder
2. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html

```
12
13      // Dump the relative path to the file
14      var_dump($file->getRelativePathname());
15  }
```

The `$file` is an instance of *SplFileInfo*[3] which extends *SplFileInfo*[4] to provide methods to work with relative paths.

The above code prints the names of all the files in the current directory recursively. The Finder class uses a fluent interface, so all methods return the Finder instance.

A Finder instance is a PHP *Iterator*[5]. So, instead of iterating over the Finder with `foreach`, you can also convert it to an array with the *iterator_to_array*[6] method, or get the number of items with *iterator_count*[7].

When searching through multiple locations passed to the *in()*[8] method, a separate iterator is created internally for every location. This means we have multiple result sets aggregated into one. Since *iterator_to_array*[9] uses keys of result sets by default, when converting to an array, some keys might be duplicated and their values overwritten. This can be avoided by passing `false` as a second parameter to *iterator_to_array*[10].

Criteria

There are lots of ways to filter and sort your results.

Location

The location is the only mandatory criteria. It tells the finder which directory to use for the search:

Listing 60-2
```
$finder->in(__DIR__);
```

Search in several locations by chaining calls to *in()*[11]:

Listing 60-3
```
$finder->files()->in(__DIR__)->in('/elsewhere');
```

Use wildcard characters to search in the directories matching a pattern:

Listing 60-4
```
$finder->in('src/Symfony/*/*/Resources');
```

Each pattern has to resolve to at least one directory path.

Exclude directories from matching with the *exclude()*[12] method:

3. http://api.symfony.com/3.0/Symfony/Component/Finder/SplFileInfo.html
4. http://php.net/manual/en/class.splfileinfo.php
5. http://php.net/manual/en/class.iterator.php
6. http://php.net/manual/en/function.iterator-to-array.php
7. http://php.net/manual/en/function.iterator-count.php
8. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_in
9. http://php.net/manual/en/function.iterator-to-array.php
10. http://php.net/manual/en/function.iterator-to-array.php
11. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_in
12. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_exclude

```php
$finder->in(__DIR__)->exclude('ruby');
```

Listing 60-5

It's also possible to ignore directories that you don't have permission to read:

Listing 60-6
```php
$finder->ignoreUnreadableDirs()->in(__DIR__);
```

As the Finder uses PHP iterators, you can pass any URL with a supported *protocol*[13]:

Listing 60-7
```php
$finder->in('ftp://example.com/pub/');
```

And it also works with user-defined streams:

Listing 60-8
```php
1  use Symfony\Component\Finder\Finder;
2
3  $s3 = new \Zend_Service_Amazon_S3($key, $secret);
4  $s3->registerStreamWrapper('s3');
5
6  $finder = new Finder();
7  $finder->name('photos*')->size('< 100K')->date('since 1 hour ago');
8  foreach ($finder->in('s3://bucket-name') as $file) {
9      // ... do something with the file
10 }
```

Read the *Streams*[14] documentation to learn how to create your own streams.

Files or Directories

By default, the Finder returns files and directories; but the *files()*[15] and *directories()*[16] methods control that:

Listing 60-9
```php
$finder->files();

$finder->directories();
```

If you want to follow links, use the `followLinks()` method:

Listing 60-10
```php
$finder->files()->followLinks();
```

By default, the iterator ignores popular VCS files. This can be changed with the `ignoreVCS()` method:

Listing 60-11
```php
$finder->ignoreVCS(false);
```

Sorting

Sort the result by name or by type (directories first, then files):

Listing 60-12
```php
$finder->sortByName();

$finder->sortByType();
```

13. http://www.php.net/manual/en/wrappers.php
14. http://www.php.net/streams
15. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_files
16. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_directories

Notice that the `sort*` methods need to get all matching elements to do their jobs. For large iterators, it is slow.

You can also define your own sorting algorithm with `sort()` method:

Listing 60-13
```
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

File Name

Restrict files by name with the *name()*[17] method:

Listing 60-14
```
$finder->files()->name('*.php');
```

The `name()` method accepts globs, strings, or regexes:

Listing 60-15
```
$finder->files()->name('/\.php$/');
```

The `notName()` method excludes files matching a pattern:

Listing 60-16
```
$finder->files()->notName('*.rb');
```

File Contents

Restrict files by contents with the *contains()*[18] method:

Listing 60-17
```
$finder->files()->contains('lorem ipsum');
```

The `contains()` method accepts strings or regexes:

Listing 60-18
```
$finder->files()->contains('/lorem\s+ipsum$/i');
```

The `notContains()` method excludes files containing given pattern:

Listing 60-19
```
$finder->files()->notContains('dolor sit amet');
```

Path

Restrict files and directories by path with the *path()*[19] method:

Listing 60-20
```
$finder->path('some/special/dir');
```

On all platforms slash (i.e. `/`) should be used as the directory separator.

The `path()` method accepts a string or a regular expression:

Listing 60-21

17. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_name
18. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_contains
19. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_path

```
$finder->path('foo/bar');
$finder->path('/^foo\/bar/');
```

Internally, strings are converted into regular expressions by escaping slashes and adding delimiters:

Listing 60-22
```
1 dirname    ===>    /dirname/
2 a/b/c      ===>    /a\/b\/c/
```

The *notPath()*[20] method excludes files by path:

Listing 60-23
```
$finder->notPath('other/dir');
```

File Size

Restrict files by size with the *size()*[21] method:

Listing 60-24
```
$finder->files()->size('< 1.5K');
```

Restrict by a size range by chaining calls:

Listing 60-25
```
$finder->files()->size('>= 1K')->size('<= 2K');
```

The comparison operator can be any of the following: `>`, `>=`, `<`, `<=`, `==`, `!=`.

The target value may use magnitudes of kilobytes (`k`, `ki`), megabytes (`m`, `mi`), or gigabytes (`g`, `gi`). Those suffixed with an `i` use the appropriate `2**n` version in accordance with the *IEC standard*[22].

File Date

Restrict files by last modified dates with the *date()*[23] method:

Listing 60-26
```
$finder->date('since yesterday');
```

The comparison operator can be any of the following: `>`, `>=`, `<`, `<=`, `==`. You can also use `since` or `after` as an alias for `>`, and `until` or `before` as an alias for `<`.

The target value can be any date supported by the *strtotime*[24] function.

Directory Depth

By default, the Finder recursively traverse directories. Restrict the depth of traversing with *depth()*[25]:

Listing 60-27
```
$finder->depth('== 0');
$finder->depth('< 3');
```

Custom Filtering

To restrict the matching file with your own strategy, use *filter()*[26]:

Listing 60-28

20. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_notPath
21. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_size
22. http://physics.nist.gov/cuu/Units/binary.html
23. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_date
24. http://www.php.net/manual/en/datetime.formats.php
25. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_depth
26. http://api.symfony.com/3.0/Symfony/Component/Finder/Finder.html#method_filter

```
1 $filter = function (\SplFileInfo $file)
2 {
3     if (strlen($file) > 10) {
4         return false;
5     }
6 };
7
8 $finder->files()->filter($filter);
```

The `filter()` method takes a Closure as an argument. For each matching file, it is called with the file as a *SplFileInfo*[27] instance. The file is excluded from the result set if the Closure returns `false`.

Reading Contents of Returned Files

The contents of returned files can be read with *getContents()*[28]:

Listing 60-29

```
1 use Symfony\Component\Finder\Finder;
2
3 $finder = new Finder();
4 $finder->files()->in(__DIR__);
5
6 foreach ($finder as $file) {
7     $contents = $file->getContents();
8
9     // ...
10 }
```

27. http://api.symfony.com/3.0/Symfony/Component/Finder/SplFileInfo.html

28. http://api.symfony.com/3.0/Symfony/Component/Finder/SplFileInfo.html#method_getContents

Chapter 61

The Form Component

> The Form component allows you to easily create, process and reuse HTML forms.

The Form component is a tool to help you solve the problem of allowing end-users to interact with the data and modify the data in your application. And though traditionally this has been through HTML forms, the component focuses on processing data to and from your client and application, whether that data be from a normal form post or from an API.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/form` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/form*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Configuration

If you are working with the full-stack Symfony Framework, the Form component is already configured for you. In this case, skip to Creating a simple Form.

In Symfony, forms are represented by objects and these objects are built by using a *form factory*. Building a form factory is simple:

Listing 61-1

1. `https://packagist.org/packages/symfony/form`

```
use Symfony\Component\Form\Forms;

$formFactory = Forms::createFormFactory();
```

This factory can already be used to create basic forms, but it is lacking support for very important features:

- **Request Handling:** Support for request handling and file uploads;
- **CSRF Protection:** Support for protection against Cross-Site-Request-Forgery (CSRF) attacks;
- **Templating:** Integration with a templating layer that allows you to reuse HTML fragments when rendering a form;
- **Translation:** Support for translating error messages, field labels and other strings;
- **Validation:** Integration with a validation library to generate error messages for submitted data.

The Symfony Form component relies on other libraries to solve these problems. Most of the time you will use Twig and the Symfony *HttpFoundation*, Translation and Validator components, but you can replace any of these with a different library of your choice.

The following sections explain how to plug these libraries into the form factory.

For a working example, see *https://github.com/webmozart/standalone-forms*

Request Handling

To process form data, you'll need to call the *handleRequest()*[2] method:

Listing 61-2
```
$form->handleRequest();
```

Behind the scenes, this uses a *NativeRequestHandler*[3] object to read data off of the correct PHP superglobals (i.e. `$_POST` or `$_GET`) based on the HTTP method configured on the form (POST is default).

If you need more control over exactly when your form is submitted or which data is passed to it, you can use the `submit()`[4] for this. Read more about it in the cookbook.

2. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_handleRequest
3. http://api.symfony.com/3.0/Symfony/Component/Form/NativeRequestHandler.html
4. *http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_submit*

Integration with the HttpFoundation Component

If you use the HttpFoundation component, then you should add the *`HttpFoundationExtension`*[5] to your form factory:

Listing 61-3
```
use Symfony\Component\Form\Forms;
use Symfony\Component\Form\Extension\HttpFoundation\HttpFoundationExtension;

$formFactory = Forms::createFormFactoryBuilder()
    ->addExtension(new HttpFoundationExtension())
    ->getFormFactory();
```

Now, when you process a form, you can pass the *`Request`*[6] object to *`handleRequest()`*[7]:

Listing 61-4
```
$form->handleRequest($request);
```

For more information about the HttpFoundation component or how to install it, see *The HttpFoundation Component*.

CSRF Protection

Protection against CSRF attacks is built into the Form component, but you need to explicitly enable it or replace it with a custom solution. If you want to use the built-in support, require the Security CSRF component by executing `composer require symfony/security-csrf`.

The following snippet adds CSRF protection to the form factory:

Listing 61-5
```
use Symfony\Component\Form\Forms;
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\Security\Extension\Csrf\CsrfExtension;
use Symfony\Component\Security\Csrf\TokenStorage\SessionTokenStorage;
use Symfony\Component\Security\Csrf\TokenGenerator\UriSafeTokenGenerator;
use Symfony\Component\Security\Csrf\CsrfTokenManager;

// create a Session object from the HttpFoundation component
$session = new Session();

$csrfGenerator = new UriSafeTokenGenerator();
$csrfStorage = new SessionTokenStorage($session);
$csrfManager = new CsrfTokenManager($csrfGenerator, $csrfStorage);

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->addExtension(new CsrfExtension($csrfStorage))
    ->getFormFactory();
```

Internally, this extension will automatically add a hidden field to every form (called `_token` by default) whose value is automatically generated by the CSRF generator and validated when binding the form.

5. http://api.symfony.com/3.0/Symfony/Component/Form/Extension/HttpFoundation/HttpFoundationExtension.html
6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
7. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_handleRequest

If you're not using the HttpFoundation component, you can use *NativeSessionTokenStorage*[8] instead, which relies on PHP's native session handling:

Listing 61-6
```
use Symfony\Component\Security\Csrf\TokenStorage\NativeSessionTokenStorage;

$csrfStorage = new NativeSessionTokenStorage();
// ...
```

Twig Templating

If you're using the Form component to process HTML forms, you'll need a way to easily render your form as HTML form fields (complete with field values, errors, and labels). If you use *Twig*[9] as your template engine, the Form component offers a rich integration.

To use the integration, you'll need the `TwigBridge`, which provides integration between Twig and several Symfony components. If you're using Composer, you could install the latest 3.x version by adding the following `require` line to your `composer.json` file:

Listing 61-7
```
{
    "require": {
        "symfony/twig-bridge": "~3.0"
    }
}
```

The TwigBridge integration provides you with several *Twig Functions* that help you render the HTML widget, label and error for each field (as well as a few other things). To configure the integration, you'll need to bootstrap or access Twig and add the *FormExtension*[10]:

Listing 61-8
```
use Symfony\Component\Form\Forms;
use Symfony\Bridge\Twig\Extension\FormExtension;
use Symfony\Bridge\Twig\Form\TwigRenderer;
use Symfony\Bridge\Twig\Form\TwigRendererEngine;

// the Twig file that holds all the default markup for rendering forms
// this file comes with TwigBridge
$defaultFormTheme = 'form_div_layout.html.twig';

$vendorDir = realpath(__DIR__.'/../vendor');
// the path to TwigBridge library so Twig can locate the
// form_div_layout.html.twig file
$appVariableReflection = new \ReflectionClass('\Symfony\Bridge\Twig\AppVariable');
$vendorTwigBridgeDir = dirname($appVariableReflection->getFileName());
// the path to your other templates
$viewsDir = realpath(__DIR__.'/../views');

$twig = new Twig_Environment(new Twig_Loader_Filesystem(array(
    $viewsDir,
    $vendorTwigBridgeDir.'/Resources/views/Form',
)));
$formEngine = new TwigRendererEngine(array($defaultFormTheme));
$formEngine->setEnvironment($twig);
// add the FormExtension to Twig
```

8. http://api.symfony.com/3.0/Symfony/Component/Security/Csrf/TokenStorage/NativeSessionTokenStorage.html
9. http://twig.sensiolabs.org
10. http://api.symfony.com/3.0/Symfony/Bridge/Twig/Extension/FormExtension.html

```
25 $twig->addExtension(
26     new FormExtension(new TwigRenderer($formEngine, $csrfProvider))
27 );
28
29 // create your form factory as normal
30 $formFactory = Forms::createFormFactoryBuilder()
31     // ...
32     ->getFormFactory();
```

The exact details of your *Twig Configuration*[11] will vary, but the goal is always to add the *FormExtension*[12] to Twig, which gives you access to the Twig functions for rendering forms. To do this, you first need to create a *TwigRendererEngine*[13], where you define your form themes (i.e. resources/files that define form HTML markup).

For general details on rendering forms, see *How to Customize Form Rendering*.

If you use the Twig integration, read "Translation" below for details on the needed translation filters.

Translation

If you're using the Twig integration with one of the default form theme files (e.g. `form_div_layout.html.twig`), there are 2 Twig filters (`trans` and `transChoice`) that are used for translating form labels, errors, option text and other strings.

To add these Twig filters, you can either use the built-in *TranslationExtension*[14] that integrates with Symfony's Translation component, or add the 2 Twig filters yourself, via your own Twig extension.

To use the built-in integration, be sure that your project has Symfony's Translation and *Config* components installed. If you're using Composer, you could get the latest 3.x version of each of these by adding the following to your `composer.json` file:

Listing 61-9

```
1 {
2     "require": {
3         "symfony/translation": "~3.0",
4         "symfony/config": "~3.0"
5     }
6 }
```

Next, add the *TranslationExtension*[15] to your `Twig_Environment` instance:

Listing 61-10

```
1 use Symfony\Component\Form\Forms;
2 use Symfony\Component\Translation\Translator;
3 use Symfony\Component\Translation\Loader\XliffFileLoader;
4 use Symfony\Bridge\Twig\Extension\TranslationExtension;
5
6 // create the Translator
7 $translator = new Translator('en');
8 // somehow load some translations into it
```

11. http://twig.sensiolabs.org/doc/intro.html
12. http://api.symfony.com/3.0/Symfony/Bridge/Twig/Extension/FormExtension.html
13. http://api.symfony.com/3.0/Symfony/Bridge/Twig/Form/TwigRendererEngine.html
14. http://api.symfony.com/3.0/Symfony/Bridge/Twig/Extension/TranslationExtension.html
15. http://api.symfony.com/3.0/Symfony/Bridge/Twig/Extension/TranslationExtension.html

```
9  $translator->addLoader('xlf', new XliffFileLoader());
10 $translator->addResource(
11     'xlf',
12     __DIR__.'/path/to/translations/messages.en.xlf',
13     'en'
14 );
15
16 // add the TranslationExtension (gives us trans and transChoice filters)
17 $twig->addExtension(new TranslationExtension($translator));
18
19 $formFactory = Forms::createFormFactoryBuilder()
20     // ...
21     ->getFormFactory();
```

Depending on how your translations are being loaded, you can now add string keys, such as field labels, and their translations to your translation files.

For more details on translations, see *Translations*.

Validation

The Form component comes with tight (but optional) integration with Symfony's Validator component. If you're using a different solution for validation, no problem! Simply take the submitted/bound data of your form (which is an array or object) and pass it through your own validation system.

To use the integration with Symfony's Validator component, first make sure it's installed in your application. If you're using Composer and want to install the latest 3.x version, add this to your `composer.json`:

Listing 61-11
```
1 {
2     "require": {
3         "symfony/validator": "~3.0"
4     }
5 }
```

If you're not familiar with Symfony's Validator component, read more about it: *Validation*. The Form component comes with a *`ValidatorExtension`*[16] class, which automatically applies validation to your data on bind. These errors are then mapped to the correct field and rendered.

Your integration with the Validation component will look something like this:

Listing 61-12
```
1  use Symfony\Component\Form\Forms;
2  use Symfony\Component\Form\Extension\Validator\ValidatorExtension;
3  use Symfony\Component\Validator\Validation;
4
5  $vendorDir = realpath(__DIR__.'/../vendor');
6  $vendorFormDir = $vendorDir.'/symfony/form';
7  $vendorValidatorDir = $vendorDir.'/symfony/validator';
8
9  // create the validator - details will vary
10 $validator = Validation::createValidator();
11
12 // there are built-in translations for the core error messages
13 $translator->addResource(
14     'xlf',
15     $vendorFormDir.'/Resources/translations/validators.en.xlf',
```

16. http://api.symfony.com/3.0/Symfony/Component/Form/Extension/Validator/ValidatorExtension.html

```
16     'en',
17     'validators'
18 );
19 $translator->addResource(
20     'xlf',
21     $vendorValidatorDir.'/Resources/translations/validators.en.xlf',
22     'en',
23     'validators'
24 );
25
26 $formFactory = Forms::createFormFactoryBuilder()
27     // ...
28     ->addExtension(new ValidatorExtension($validator))
29     ->getFormFactory();
```

To learn more, skip down to the Form Validation section.

Accessing the Form Factory

Your application only needs one form factory, and that one factory object should be used to create any and all form objects in your application. This means that you should create it in some central, bootstrap part of your application and then access it whenever you need to build a form.

In this document, the form factory is always a local variable called `$formFactory`. The point here is that you will probably need to create this object in some more "global" way so you can access it from anywhere.

Exactly how you gain access to your one form factory is up to you. If you're using a Service Container, then you should add the form factory to your container and grab it out whenever you need to. If your application uses global or static variables (not usually a good idea), then you can store the object on some static class or do something similar.

Regardless of how you architect your application, just remember that you should only have one form factory and that you'll need to be able to access it throughout your application.

Creating a simple Form

If you're using the Symfony Framework, then the form factory is available automatically as a service called `form.factory`. Also, the default base controller class has a *`createFormBuilder()`*[17] method, which is a shortcut to fetch the form factory and call `createBuilder` on it.

Creating a form is done via a *`FormBuilder`*[18] object, where you build and configure different fields. The form builder is created from the form factory.

Listing 61-13

```
1 use Symfony\Component\Form\Extension\Core\Type\TextType;
2 use Symfony\Component\Form\Extension\Core\Type\DateType;
3
4 // ...
5
```

17. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller.html#method_createFormBuilder
18. http://api.symfony.com/3.0/Symfony/Component/Form/FormBuilder.html

```
6  $form = $formFactory->createBuilder()
7      ->add('task', TextType::class)
8      ->add('dueDate', DateType::class)
9      ->getForm();
10
11 var_dump($twig->render('new.html.twig', array(
12     'form' => $form->createView(),
13 )));
```

As you can see, creating a form is like writing a recipe: you call `add` for each new field you want to create. The first argument to `add` is the name of your field, and the second is the fully qualified class name. The Form component comes with a lot of *built-in types*.

Now that you've built your form, learn how to render it and process the form submission.

Setting default Values

If you need your form to load with some default values (or you're building an "edit" form), simply pass in the default data when creating your form builder:

Listing 61-14

```
1  use Symfony\Component\Form\Extension\Core\Type\FormType;
2  use Symfony\Component\Form\Extension\Core\Type\TextType;
3  use Symfony\Component\Form\Extension\Core\Type\DateType;
4
5  // ...
6
7  $defaults = array(
8      'dueDate' => new \DateTime('tomorrow'),
9  );
10
11 $form = $formFactory->createBuilder(FormType::class, $defaults)
12     ->add('task', TextType::class)
13     ->add('dueDate', DateType::class)
14     ->getForm();
```

In this example, the default data is an array. Later, when you use the data_class option to bind data directly to objects, your default data will be an instance of that object.

Rendering the Form

Now that the form has been created, the next step is to render it. This is done by passing a special form "view" object to your template (notice the `$form->createView()` in the controller above) and using a set of form helper functions:

Listing 61-15

```
1 {{ form_start(form) }}
2     {{ form_widget(form) }}
3
4     <input type="submit" />
5 {{ form_end(form) }}
```

That's it! By printing `form_widget(form)`, each field in the form is rendered, along with a label and error message (if there is one). As easy as this is, it's not very flexible (yet). Usually, you'll want to render each form field individually so you can control how the form looks. You'll learn how to do that in the "Rendering a Form in a Template" section.

Changing a Form's Method and Action

By default, a form is submitted to the same URI that rendered the form with an HTTP POST request. This behavior can be changed using the action and method options (the `method` option is also used by `handleRequest()` to determine whether a form has been submitted):

Listing 61-16

```
use Symfony\Component\Form\Extension\Core\Type\FormType;

// ...

$formBuilder = $formFactory->createBuilder(FormType::class, null, array(
    'action' => '/search',
    'method' => 'GET',
));

// ...
```

Handling Form Submissions

To handle form submissions, use the *`handleRequest()`*[19] method:

Listing 61-17

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\DateType;

// ...

$form = $formFactory->createBuilder()
    ->add('task', TextType::class)
    ->add('dueDate', DateType::class)
    ->getForm();

$request = Request::createFromGlobals();

$form->handleRequest($request);

if ($form->isValid()) {
```

19. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_handleRequest

```
18     $data = $form->getData();
19
20     // ... perform some action, such as saving the data to the database
21
22     $response = new RedirectResponse('/task/success');
23     $response->prepare($request);
24
25     return $response->send();
26 }
27
28 // ...
```

This defines a common form "workflow", which contains 3 different possibilities:

1. On the initial GET request (i.e. when the user "surfs" to your page), build your form and render it;

If the request is a POST, process the submitted data (via `handleRequest()`). Then:

2. if the form is invalid, re-render the form (which will now contain errors);
3. if the form is valid, perform some action and redirect.

Luckily, you don't need to decide whether or not a form has been submitted. Just pass the current request to the `handleRequest()` method. Then, the Form component will do all the necessary work for you.

Form Validation

The easiest way to add validation to your form is via the `constraints` option when building each field:

Listing 61-18

```
1  use Symfony\Component\Validator\Constraints\NotBlank;
2  use Symfony\Component\Validator\Constraints\Type;
3  use Symfony\Component\Form\Extension\Core\Type\TextType;
4  use Symfony\Component\Form\Extension\Core\Type\DateType;
5
6  $form = $formFactory->createBuilder()
7      ->add('task', TextType::class, array(
8          'constraints' => new NotBlank(),
9      ))
10     ->add('dueDate', DateType::class, array(
11         'constraints' => array(
12             new NotBlank(),
13             new Type('\DateTime'),
14         )
15     ))
16     ->getForm();
```

When the form is bound, these validation constraints will be applied automatically and the errors will display next to the fields on error.

For a list of all of the built-in validation constraints, see *Validation Constraints Reference*.

Accessing Form Errors

You can use the *getErrors()*[20] method to access the list of errors. It returns a *FormErrorIterator*[21] instance:

Listing 61-19

```
$form = ...;

// ...

// a FormErrorIterator instance, but only errors attached to this
// form level (e.g. "global errors)
$errors = $form->getErrors();

// a FormErrorIterator instance, but only errors attached to the
// "firstName" field
$errors = $form['firstName']->getErrors();

// a FormErrorIterator instance in a flattened structure
// use getOrigin() to determine the form causing the error
$errors = $form->getErrors(true);

// a FormErrorIterator instance representing the form tree structure
$errors = $form->getErrors(true, false);
```

20. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_getErrors
21. http://api.symfony.com/3.0/Symfony/Component/Form/FormErrorIterator.html

Chapter 62

Creating a custom Type Guesser

The Form component can guess the type and some options of a form field by using type guessers. The component already includes a type guesser using the assertions of the Validation component, but you can also add your own custom type guessers.

Form Type Guessers in the Bridges

Symfony also provides some form type guessers in the bridges:

- *PropelTypeGuesser*[1] provided by the Propel1 bridge;
- *DoctrineOrmTypeGuesser*[2] provided by the Doctrine bridge.

Create a PHPDoc Type Guesser

In this section, you are going to build a guesser that reads information about fields from the PHPDoc of the properties. At first, you need to create a class which implements *FormTypeGuesserInterface*[3]. This interface requires 4 methods:

- *guessType()*[4] - tries to guess the type of a field;
- *guessRequired()*[5] - tries to guess the value of the required option;
- *guessMaxLength()*[6] - tries to guess the value of the `maxlength` input attribute;
- *guessPattern()*[7] - tries to guess the value of the `pattern` input attribute.

Start by creating the class and these methods. Next, you'll learn how to fill each on.

Listing 62-1

1. http://api.symfony.com/3.0/Symfony/Bridge/Propel1/Form/PropelTypeGuesser.html
2. http://api.symfony.com/3.0/Symfony/Bridge/Doctrine/Form/DoctrineOrmTypeGuesser.html
3. http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeGuesserInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeGuesserInterface.html#method_guessType
5. http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeGuesserInterface.html#method_guessRequired
6. http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeGuesserInterface.html#method_guessMaxLength
7. http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeGuesserInterface.html#method_guessPattern

```php
namespace Acme\Form;

use Symfony\Component\Form\FormTypeGuesserInterface;

class PHPDocTypeGuesser implements FormTypeGuesserInterface
{
    public function guessType($class, $property)
    {
    }

    public function guessRequired($class, $property)
    {
    }

    public function guessMaxLength($class, $property)
    {
    }

    public function guessPattern($class, $property)
    {
    }
}
```

Guessing the Type

When guessing a type, the method returns either an instance of *TypeGuess*[8] or nothing, to determine that the type guesser cannot guess the type.

The `TypeGuess` constructor requires 3 options:

- The type name (one of the *form types*);
- Additional options (for instance, when the type is `entity`, you also want to set the `class` option). If no types are guessed, this should be set to an empty array;
- The confidence that the guessed type is correct. This can be one of the constants of the *Guess*[9] class: `LOW_CONFIDENCE`, `MEDIUM_CONFIDENCE`, `HIGH_CONFIDENCE`, `VERY_HIGH_CONFIDENCE`. After all type guessers have been executed, the type with the highest confidence is used.

With this knowledge, you can easily implement the `guessType` method of the `PHPDocTypeGuesser`:

Listing 62-2

```php
namespace Acme\Form;

use Symfony\Component\Form\Guess\Guess;
use Symfony\Component\Form\Guess\TypeGuess;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\NumberType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;

class PHPDocTypeGuesser implements FormTypeGuesserInterface
{
    public function guessType($class, $property)
    {
        $annotations = $this->readPhpDocAnnotations($class, $property);

```

8. http://api.symfony.com/3.0/Symfony/Component/Form/Guess/TypeGuess.html
9. http://api.symfony.com/3.0/Symfony/Component/Form/Guess/Guess.html

```
16         if (!isset($annotations['var'])) {
17             return; // guess nothing if the @var annotation is not available
18         }
19
20         // otherwise, base the type on the @var annotation
21         switch ($annotations['var']) {
22             case 'string':
23                 // there is a high confidence that the type is text when
24                 // @var string is used
25                 return new TypeGuess(TextType::class, array(), Guess::HIGH_CONFIDENCE);
26
27             case 'int':
28             case 'integer':
29                 // integers can also be the id of an entity or a checkbox (0 or 1)
30                 return new TypeGuess(IntegerType::class, array(),
31 Guess::MEDIUM_CONFIDENCE);
32
33             case 'float':
34             case 'double':
35             case 'real':
36                 return new TypeGuess(NumberType::class, array(), Guess::MEDIUM_CONFIDENCE);
37
38             case 'boolean':
39             case 'bool':
40                 return new TypeGuess(CheckboxType::class, array(), Guess::HIGH_CONFIDENCE);
41
42             default:
43                 // there is a very low confidence that this one is correct
44                 return new TypeGuess(TextType::class, array(), Guess::LOW_CONFIDENCE);
45         }
46     }
47
48     protected function readPhpDocAnnotations($class, $property)
49     {
50         $reflectionProperty = new \ReflectionProperty($class, $property);
51         $phpdoc = $reflectionProperty->getDocComment();
52
53         // parse the $phpdoc into an array like:
54         // array('type' => 'string', 'since' => '1.0')
55         $phpdocTags = ...;
56
57         return $phpdocTags;
58     }
   }
```

This type guesser can now guess the field type for a property if it has PHPdoc!

Guessing Field Options

The other 3 methods (`guessMaxLength`, `guessRequired` and `guessPattern`) return a *ValueGuess*[10] instance with the value of the option. This constructor has 2 arguments:

- The value of the option;
- The confidence that the guessed value is correct (using the constants of the `Guess` class).

`null` is guessed when you believe the value of the option should not be set.

10. http://api.symfony.com/3.0/Symfony/Component/Form/Guess/ValueGuess.html

You should be very careful using the `guessPattern` method. When the type is a float, you cannot use it to determine a min or max value of the float (e.g. you want a float to be greater than `5`, `4.512313` is not valid but `length(4.512314) > length(5)` is, so the pattern will succeed). In this case, the value should be set to `null` with a `MEDIUM_CONFIDENCE`.

Registering a Type Guesser

The last thing you need to do is registering your custom type guesser by using *addTypeGuesser()*[11] or *addTypeGuessers()*[12]:

Listing 62-3

```php
use Symfony\Component\Form\Forms;
use Acme\Form\PHPDocTypeGuesser;

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->addTypeGuesser(new PHPDocTypeGuesser())
    ->getFormFactory();

// ...
```

When you use the Symfony Framework, you need to register your type guesser and tag it with `form.type_guesser`. For more information see the tag reference.

11. http://api.symfony.com/3.0/Symfony/Component/Form/FormFactoryBuilder.html#method_addTypeGuesser
12. http://api.symfony.com/3.0/Symfony/Component/Form/FormFactoryBuilder.html#method_addTypeGuessers

Chapter 63
Form Events

The Form component provides a structured process to let you customize your forms, by making use of the *EventDispatcher component*. Using form events, you may modify information or fields at different steps of the workflow: from the population of the form to the submission of the data from the request.

Registering an event listener is very easy using the Form component.

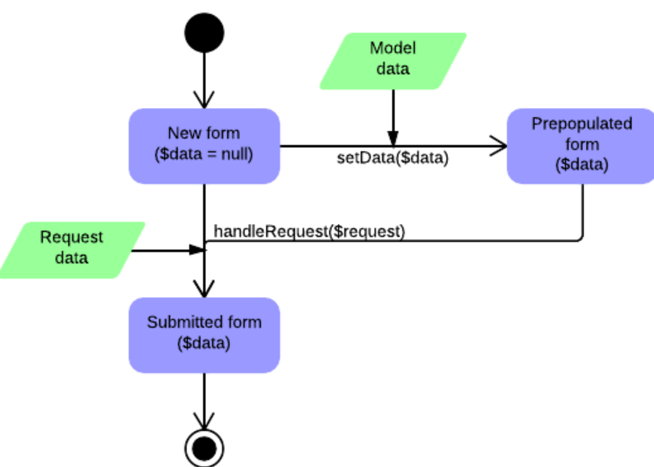
For example, if you wish to register a function to the `FormEvents::PRE_SUBMIT` event, the following code lets you add a field, depending on the request values:

Listing 63-1

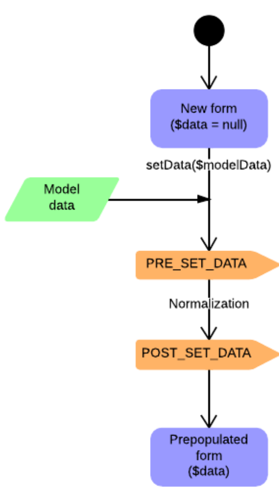
```
// ...

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;

$listener = function (FormEvent $event) {
    // ...
};

$form = $formFactory->createBuilder()
    // add form fields
    ->addEventListener(FormEvents::PRE_SUBMIT, $listener);

// ...
```

The Form Workflow

The Form Submission Workflow



1) Pre-populating the Form (`FormEvents::PRE_SET_DATA` and `FormEvents::POST_SET_DATA`)



Two events are dispatched during pre-population of a form, when *`Form::setData()`*[1] is called: `FormEvents::PRE_SET_DATA` and `FormEvents::POST_SET_DATA`.

A) The `FormEvents::PRE_SET_DATA` Event

The `FormEvents::PRE_SET_DATA` event is dispatched at the beginning of the `Form::setData()` method. It can be used to:

- Modify the data given during pre-population;
- Modify a form depending on the pre-populated data (adding or removing fields dynamically).

1. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_setData

Data Type	Value
Model data	`null`
Normalized data	`null`
View data	`null`

See all form events at a glance in the Form Events Information Table.

During `FormEvents::PRE_SET_DATA`, *`Form::setData()`*[2] is locked and will throw an exception if used. If you wish to modify data, you should use *`FormEvent::setData()`*[3] instead.

`FormEvents::PRE_SET_DATA` in the Form component

The `collection` form type relies on the `Symfony\Component\Form\Extension\Core\EventListener\ResizeFormListener` subscriber, listening to the `FormEvents::PRE_SET_DATA` event in order to reorder the form's fields depending on the data from the pre-populated object, by removing and adding all form rows.

B) The `FormEvents::POST_SET_DATA` Event

The `FormEvents::POST_SET_DATA` event is dispatched at the end of the *`Form::setData()`*[4] method. This event is mostly here for reading data after having pre-populated the form.

Data Type	Value
Model data	Model data injected into `setData()`
Normalized data	Model data transformed using a model transformer
View data	Normalized data transformed using a view transformer

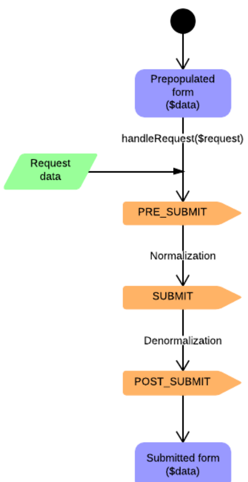
See all form events at a glance in the Form Events Information Table.

`FormEvents::POST_SET_DATA` in the Form component

The `Symfony\Component\Form\Extension\DataCollector\EventListener\DataCollectorListener` class is subscribed to listen to the `FormEvents::POST_SET_DATA` event in order to collect information about the forms from the denormalized model and view data.

2. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_setData
3. http://api.symfony.com/3.0/Symfony/Component/Form/FormEvent.html#method_setData
4. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_setData

2) Submitting a Form (`FormEvents::PRE_SUBMIT`, `FormEvents::SUBMIT` and `FormEvents::POST_SUBMIT`)



Three events are dispatched when *Form::handleRequest()*[5] or *Form::submit()*[6] are called: `FormEvents::PRE_SUBMIT`, `FormEvents::SUBMIT`, `FormEvents::POST_SUBMIT`.

A) The `FormEvents::PRE_SUBMIT` Event

The `FormEvents::PRE_SUBMIT` event is dispatched at the beginning of the *Form::submit()*[7] method.

It can be used to:

- Change data from the request, before submitting the data to the form;
- Add or remove form fields, before submitting the data to the form.

Data Type	Value
Model data	Same as in `FormEvents::POST_SET_DATA`
Normalized data	Same as in `FormEvents::POST_SET_DATA`
View data	Same as in `FormEvents::POST_SET_DATA`

See all form events at a glance in the Form Events Information Table.

`FormEvents::PRE_SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\Core\EventListener\TrimListener` subscriber subscribes to the `FormEvents::PRE_SUBMIT` event in order to trim the request's data (for string values). The `Symfony\Component\Form\Extension\Csrf\EventListener\CsrfValidationListener` subscriber subscribes to the `FormEvents::PRE_SUBMIT` event in order to validate the CSRF token.

5. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_handleRequest
6. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_submit
7. http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_submit

B) The `FormEvents::SUBMIT` Event

The `FormEvents::SUBMIT` event is dispatched just before the *`Form::submit()`*[8] method transforms back the normalized data to the model and view data.

It can be used to change data from the normalized representation of the data.

Data Type	Value
Model data	Same as in `FormEvents::POST_SET_DATA`
Normalized data	Data from the request reverse-transformed from the request using a view transformer
View data	Same as in `FormEvents::POST_SET_DATA`

See all form events at a glance in the Form Events Information Table.

At this point, you cannot add or remove fields to the form.

`FormEvents::SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\Core\EventListener\FixUrlProtocolListener` subscribes to the `FormEvents::SUBMIT` event in order to prepend a default protocol to URL fields that were submitted without a protocol.

C) The `FormEvents::POST_SUBMIT` Event

The `FormEvents::POST_SUBMIT` event is dispatched after the *`Form::submit()`*[9] once the model and view data have been denormalized.

It can be used to fetch data after denormalization.

Data Type	Value
Model data	Normalized data reverse-transformed using a model transformer
Normalized data	Same as in `FormEvents::SUBMIT`
View data	Normalized data transformed using a view transformer

See all form events at a glance in the Form Events Information Table.

At this point, you cannot add or remove fields to the form.

8. `http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_submit`
9. `http://api.symfony.com/3.0/Symfony/Component/Form/Form.html#method_submit`

`FormEvents::POST_SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\DataCollector\EventListener\DataCollectorListener` subscribes to the `FormEvents::POST_SUBMIT` event in order to collect information about the forms. The `Symfony\Component\Form\Extension\Validator\EventListener\ValidationListener` subscribes to the `FormEvents::POST_SUBMIT` event in order to automatically validate the denormalized object and to update the normalized representation as well as the view representations.

Registering Event Listeners or Event Subscribers

In order to be able to use Form events, you need to create an event listener or an event subscriber and register it to an event.

The name of each of the "form" events is defined as a constant on the *`FormEvents`*[10] class. Additionally, each event callback (listener or subscriber method) is passed a single argument, which is an instance of *`FormEvent`*[11]. The event object contains a reference to the current state of the form and the current data being processed.

Name	**FormEvents Constant**	**Event's Data**
`form.pre_set_data`	`FormEvents::PRE_SET_DATA`	Model data
`form.post_set_data`	`FormEvents::POST_SET_DATA`	Model data
`form.pre_bind`	`FormEvents::PRE_SUBMIT`	Request data
`form.bind`	`FormEvents::SUBMIT`	Normalized data
`form.post_bind`	`FormEvents::POST_SUBMIT`	View data

Event Listeners

An event listener may be any type of valid callable.

Creating and binding an event listener to the form is very easy:

Listing 63-2

```
// ...

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;

$form = $formFactory->createBuilder()
    ->add('username', TextType::class)
    ->add('show_email', CheckboxType::class)
    ->addEventListener(FormEvents::PRE_SUBMIT, function (FormEvent $event) {
        $user = $event->getData();
        $form = $event->getForm();

```

10. http://api.symfony.com/3.0/Symfony/Component/Form/FormEvents.html
11. http://api.symfony.com/3.0/Symfony/Component/Form/FormEvent.html

```
16            if (!$user) {
17                return;
18            }
19
20            // Check whether the user has chosen to display his email or not.
21            // If the data was submitted previously, the additional value that is
22            // included in the request variables needs to be removed.
23            if (true === $user['show_email']) {
24                $form->add('email', EmailType::class);
25            } else {
26                unset($user['email']);
27                $event->setData($user);
28            }
29        })
30        ->getForm();
31
32    // ...
```

When you have created a form type class, you can use one of its methods as a callback for better readability:

Listing 63-3

```
1  use Symfony\Component\Form\Extension\Core\Type\TextType;
2  use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
3
4  // ...
5
6  class SubscriptionType extends AbstractType
7  {
8      public function buildForm(FormBuilderInterface $builder, array $options)
9      {
10         $builder->add('username', TextType::class);
11         $builder->add('show_email', CheckboxType::class);
12         $builder->addEventListener(
13             FormEvents::PRE_SET_DATA,
14             array($this, 'onPreSetData')
15         );
16     }
17
18     public function onPreSetData(FormEvent $event)
19     {
20         // ...
21     }
22 }
```

Event Subscribers

Event subscribers have different uses:

- Improving readability;
- Listening to multiple events;
- Regrouping multiple listeners inside a single class.

Listing 63-4

```
1  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
2  use Symfony\Component\Form\FormEvent;
3  use Symfony\Component\Form\FormEvents;
```

```
4 use Symfony\Component\Form\Extension\Core\Type\EmailType;
5
6 class AddEmailFieldListener implements EventSubscriberInterface
7 {
8     public static function getSubscribedEvents()
9     {
10        return array(
11            FormEvents::PRE_SET_DATA => 'onPreSetData',
12            FormEvents::PRE_SUBMIT   => 'onPreSubmit',
13        );
14    }
15
16    public function onPreSetData(FormEvent $event)
17    {
18        $user = $event->getData();
19        $form = $event->getForm();
20
21        // Check whether the user from the initial data has chosen to
22        // display his email or not.
23        if (true === $user->isShowEmail()) {
24            $form->add('email', EmailType::class);
25        }
26    }
27
28    public function onPreSubmit(FormEvent $event)
29    {
30        $user = $event->getData();
31        $form = $event->getForm();
32
33        if (!$user) {
34            return;
35        }
36
37        // Check whether the user has chosen to display his email or not.
38        // If the data was submitted previously, the additional value that
39        // is included in the request variables needs to be removed.
40        if (true === $user['show_email']) {
41            $form->add('email', EmailType::class);
42        } else {
43            unset($user['email']);
44            $event->setData($user);
45        }
46    }
47 }
```

To register the event subscriber, use the addEventSubscriber() method:

Listing 63-5

```
1 use Symfony\Component\Form\Extension\Core\Type\TextType;
2 use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
3
4 // ...
5
6 $form = $formFactory->createBuilder()
7     ->add('username', TextType::class)
8     ->add('show_email', CheckboxType::class)
9     ->addEventSubscriber(new AddEmailFieldListener())
10    ->getForm();
```

```
11
12 // ...
```

Chapter 64
The HttpFoundation Component

> The HttpFoundation component defines an object-oriented layer for the HTTP specification.

In PHP, the request is represented by some global variables (`$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION`, ...) and the response is generated by some functions (`echo`, `header`, `setcookie`, ...).

The Symfony HttpFoundation component replaces these default PHP global variables and functions by an object-oriented layer.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/http-foundation` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/http-foundation*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Request

The most common way to create a request is to base it on the current PHP global variables with *createFromGlobals()*[2]:

Listing 64-1
```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

which is almost equivalent to the more verbose, but also more flexible, *__construct()*[3] call:

1. https://packagist.org/packages/symfony/http-foundation
2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_createFromGlobals

Listing 64-2

```
$request = new Request(
    $_GET,
    $_POST,
    array(),
    $_COOKIE,
    $_FILES,
    $_SERVER
);
```

Accessing Request Data

A Request object holds information about the client request. This information can be accessed via several public properties:

- `request`: equivalent of `$_POST`;
- `query`: equivalent of `$_GET` (`$request->query->get('name')`);
- `cookies`: equivalent of `$_COOKIE`;
- `attributes`: no equivalent - used by your app to store other data (see below);
- `files`: equivalent of `$_FILES`;
- `server`: equivalent of `$_SERVER`;
- `headers`: mostly equivalent to a subset of `$_SERVER` (`$request->headers->get('User-Agent')`).

Each property is a *ParameterBag*[4] instance (or a sub-class of), which is a data holder class:

- `request`: *ParameterBag*[5];
- `query`: *ParameterBag*[6];
- `cookies`: *ParameterBag*[7];
- `attributes`: *ParameterBag*[8];
- `files`: *FileBag*[9];
- `server`: *ServerBag*[10];
- `headers`: *HeaderBag*[11].

All *ParameterBag*[12] instances have methods to retrieve and update their data:

all()[13]
: Returns the parameters.

keys()[14]
: Returns the parameter keys.

replace()[15]
: Replaces the current parameters by a new set.

3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method___construct
4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
7. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
9. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/FileBag.html
10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ServerBag.html
11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/HeaderBag.html
12. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_all
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_keys
15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_replace

add()[16]
: Adds parameters.

get()[17]
: Returns a parameter by name.

set()[18]
: Sets a parameter by name.

has()[19]
: Returns `true` if the parameter is defined.

remove()[20]
: Removes a parameter.

The *ParameterBag*[21] instance also has some methods to filter the input values:

getAlpha()[22]
: Returns the alphabetic characters of the parameter value;

getAlnum()[23]
: Returns the alphabetic characters and digits of the parameter value;

getBoolean()[24]
: Returns the parameter value converted to boolean;

  New in version 2.6: The `getBoolean()` method was introduced in Symfony 2.6.

getDigits()[25]
: Returns the digits of the parameter value;

getInt()[26]
: Returns the parameter value converted to integer;

filter()[27]
: Filters the parameter by using the PHP *filter_var*[28] function.

All getters take up to two arguments: the first one is the parameter name and the second one is the default value to return if the parameter does not exist:

Listing 64-3

```
// the query string is '?foo=bar'

$request->query->get('foo');
// returns 'bar'

$request->query->get('bar');
```

16. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_add
17. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_get
18. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_set
19. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_has
20. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_remove
21. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
22. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_getAlpha
23. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_getAlnum
24. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_getBoolean
25. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_getDigits
26. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_getInt
27. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_filter
28. http://php.net/manual/en/function.filter-var.php

```
7 // returns null
8
9 $request->query->get('bar', 'baz');
10 // returns 'baz'
```

When PHP imports the request query, it handles request parameters like `foo[bar]=bar` in a special way as it creates an array. So you can get the `foo` parameter and you will get back an array with a `bar` element:

Listing 64-4
```
1 // the query string is '?foo[bar]=baz'
2
3 $request->query->get('foo');
4 // returns array('bar' => 'baz')
5
6 $request->query->get('foo[bar]');
7 // returns null
8
9 $request->query->get('foo')['bar'];
10 // returns 'baz'
```

Thanks to the public `attributes` property, you can store additional data in the request, which is also an instance of *ParameterBag*[29]. This is mostly used to attach information that belongs to the Request and that needs to be accessed from many different points in your application. For information on how this is used in the Symfony Framework, see the Symfony book.

Finally, the raw data sent with the request body can be accessed using *getContent()*[30]:

Listing 64-5
```
$content = $request->getContent();
```

For instance, this may be useful to process a JSON string sent to the application by a remote service using the HTTP POST method.

Identifying a Request

In your application, you need a way to identify a request; most of the time, this is done via the "path info" of the request, which can be accessed via the *getPathInfo()*[31] method:

Listing 64-6
```
// for a request to http://example.com/blog/index.php/post/hello-world
// the path info is "/post/hello-world"
$request->getPathInfo();
```

Simulating a Request

Instead of creating a request based on the PHP globals, you can also simulate a request:

Listing 64-7
```
1 $request = Request::create(
2     '/hello-world',
3     'GET',
4     array('name' => 'Fabien')
5 );
```

29. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
30. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getContent
31. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getPathInfo

The *create()*[32] method creates a request based on a URI, a method and some parameters (the query parameters or the request ones depending on the HTTP method); and of course, you can also override all other variables as well (by default, Symfony creates sensible defaults for all the PHP global variables).

Based on such a request, you can override the PHP global variables via *overrideGlobals()*[33]:

Listing 64-8
```
$request->overrideGlobals();
```

You can also duplicate an existing request via *duplicate()*[34] or change a bunch of parameters with a single call to *initialize()*[35].

Accessing the Session

If you have a session attached to the request, you can access it via the *getSession()*[36] method; the *hasPreviousSession()*[37] method tells you if the request contains a session which was started in one of the previous requests.

Accessing *Accept-** Headers Data

You can easily access basic data extracted from `Accept-*` headers by using the following methods:

getAcceptableContentTypes()[38]
: Returns the list of accepted content types ordered by descending quality.

getLanguages()[39]
: Returns the list of accepted languages ordered by descending quality.

getCharsets()[40]
: Returns the list of accepted charsets ordered by descending quality.

getEncodings()[41]
: Returns the list of accepted encodings ordered by descending quality.

If you need to get full access to parsed data from `Accept`, `Accept-Language`, `Accept-Charset` or `Accept-Encoding`, you can use *AcceptHeader*[42] utility class:

Listing 64-9
```
1 use Symfony\Component\HttpFoundation\AcceptHeader;
2
3 $accept = AcceptHeader::fromString($request->headers->get('Accept'));
4 if ($accept->has('text/html')) {
5     $item = $accept->get('text/html');
6     $charset = $item->getAttribute('charset', 'utf-8');
7     $quality = $item->getQuality();
8 }
```

32. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_create
33. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_overrideGlobals
34. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_duplicate
35. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_initialize
36. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getSession
37. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_hasPreviousSession
38. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getAcceptableContentTypes
39. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getLanguages
40. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getCharsets
41. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getEncodings
42. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/AcceptHeader.html

```
9
10 // Accept header items are sorted by descending quality
11 $accepts = AcceptHeader::fromString($request->headers->get('Accept'))
12     ->all();
```

Accessing other Data

The `Request` class has many other methods that you can use to access the request information. Have a look at *the Request API*[43] for more information about them.

Overriding the Request

The `Request` class should not be overridden as it is a data object that represents an HTTP message. But when moving from a legacy system, adding methods or changing some default behavior might help. In that case, register a PHP callable that is able to create an instance of your `Request` class:

Listing 64-10

```
1  use Symfony\Component\HttpFoundation\Request;
2
3  Request::setFactory(function (
4      array $query = array(),
5      array $request = array(),
6      array $attributes = array(),
7      array $cookies = array(),
8      array $files = array(),
9      array $server = array(),
10     $content = null
11 ) {
12     return SpecialRequest::create(
13         $query,
14         $request,
15         $attributes,
16         $cookies,
17         $files,
18         $server,
19         $content
20     );
21 });
22
23 $request = Request::createFromGlobals();
```

Response

A *Response*[44] object holds all the information that needs to be sent back to the client from a given request. The constructor takes up to three arguments: the response content, the status code, and an array of HTTP headers:

Listing 64-11

```
1 use Symfony\Component\HttpFoundation\Response;
2
3 $response = new Response(
4     'Content',
```

43. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
44. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html

```
5     Response::HTTP_OK,
6     array('content-type' => 'text/html')
7 );
```

This information can also be manipulated after the Response object creation:

Listing 64-12
```
1 $response->setContent('Hello World');
2
3 // the headers public attribute is a ResponseHeaderBag
4 $response->headers->set('Content-Type', 'text/plain');
5
6 $response->setStatusCode(Response::HTTP_NOT_FOUND);
```

When setting the `Content-Type` of the Response, you can set the charset, but it is better to set it via the *`setCharset()`*[45] method:

Listing 64-13
```
$response->setCharset('ISO-8859-1');
```

Note that by default, Symfony assumes that your Responses are encoded in UTF-8.

Sending the Response

Before sending the Response, you can ensure that it is compliant with the HTTP specification by calling the *`prepare()`*[46] method:

Listing 64-14
```
$response->prepare($request);
```

Sending the response to the client is then as simple as calling *`send()`*[47]:

Listing 64-15
```
$response->send();
```

Setting Cookies

The response cookies can be manipulated through the `headers` public attribute:

Listing 64-16
```
use Symfony\Component\HttpFoundation\Cookie;

$response->headers->setCookie(new Cookie('foo', 'bar'));
```

The *`setCookie()`*[48] method takes an instance of *`Cookie`*[49] as an argument.

You can clear a cookie via the *`clearCookie()`*[50] method.

Managing the HTTP Cache

The *`Response`*[51] class has a rich set of methods to manipulate the HTTP headers related to the cache:

- *`setPublic()`*[52];

45. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setCharset
46. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_prepare
47. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_send
48. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#method_setCookie
49. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Cookie.html
50. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#method_clearCookie
51. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html
52. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setPublic

- *setPrivate()*[53];
- *expire()*[54];
- *setExpires()*[55];
- *setMaxAge()*[56];
- *setSharedMaxAge()*[57];
- *setTtl()*[58];
- *setClientTtl()*[59];
- *setLastModified()*[60];
- *setEtag()*[61];
- *setVary()*[62];

The *setCache()*[63] method can be used to set the most commonly used cache information in one method call:

Listing 64-17
```
1 $response->setCache(array(
2     'etag'          => 'abcdef',
3     'last_modified' => new \DateTime(),
4     'max_age'       => 600,
5     's_maxage'      => 600,
6     'private'       => false,
7     'public'        => true,
8 ));
```

To check if the Response validators (`ETag`, `Last-Modified`) match a conditional value specified in the client Request, use the *isNotModified()*[64] method:

Listing 64-18
```
if ($response->isNotModified($request)) {
    $response->send();
}
```

If the Response is not modified, it sets the status code to 304 and removes the actual response content.

Redirecting the User

To redirect the client to another URL, you can use the *RedirectResponse*[65] class:

Listing 64-19
```
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://example.com/');
```

53. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setPrivate
54. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_expire
55. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setExpires
56. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setMaxAge
57. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setSharedMaxAge
58. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setTtl
59. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setClientTtl
60. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setLastModified
61. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setEtag
62. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setVary
63. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setCache
64. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_isNotModified
65. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RedirectResponse.html

Streaming a Response

The *StreamedResponse*[66] class allows you to stream the Response back to the client. The response content is represented by a PHP callable instead of a string:

Listing 64-20
```
use Symfony\Component\HttpFoundation\StreamedResponse;

$response = new StreamedResponse();
$response->setCallback(function () {
    var_dump('Hello World');
    flush();
    sleep(2);
    var_dump('Hello World');
    flush();
});
$response->send();
```

The `flush()` function does not flush buffering. If `ob_start()` has been called before or the `output_buffering php.ini` option is enabled, you must call `ob_flush()` before `flush()`.

Additionally, PHP isn't the only layer that can buffer output. Your web server might also buffer based on its configuration. What's more, if you use FastCGI, buffering can't be disabled at all.

Serving Files

When sending a file, you must add a `Content-Disposition` header to your response. While creating this header for basic file downloads is easy, using non-ASCII filenames is more involving. The *makeDisposition()*[67] abstracts the hard work behind a simple API:

Listing 64-21
```
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

$d = $response->headers->makeDisposition(
    ResponseHeaderBag::DISPOSITION_ATTACHMENT,
    'foo.pdf'
);

$response->headers->set('Content-Disposition', $d);
```

Alternatively, if you are serving a static file, you can use a *BinaryFileResponse*[68]:

Listing 64-22
```
use Symfony\Component\HttpFoundation\BinaryFileResponse;

$file = 'path/to/file.txt';
$response = new BinaryFileResponse($file);
```

The `BinaryFileResponse` will automatically handle `Range` and `If-Range` headers from the request. It also supports `X-Sendfile` (see for *Nginx*[69] and *Apache*[70]). To make use of it, you need to determine whether or not the `X-Sendfile-Type` header should be trusted and call *trustXSendfileTypeHeader()*[71] if it should:

66. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/StreamedResponse.html
67. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#method_makeDisposition
68. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/BinaryFileResponse.html
69. http://wiki.nginx.org/XSendfile
70. https://tn123.org/mod_xsendfile/
71. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/BinaryFileResponse.html#method_trustXSendfileTypeHeader

```
BinaryFileResponse::trustXSendfileTypeHeader();
```

Listing 64-23

With the `BinaryFileResponse`, you can still set the `Content-Type` of the sent file, or change its `Content-Disposition`:

Listing 64-24
```
1 $response->headers->set('Content-Type', 'text/plain');
2 $response->setContentDisposition(
3     ResponseHeaderBag::DISPOSITION_ATTACHMENT,
4     'filename.txt'
5 );
```

It is possible to delete the file after the request is sent with the *deleteFileAfterSend()*[72] method. Please note that this will not work when the `X-Sendfile` header is set.

If you *just* created the file during this same request, the file *may* be sent without any content. This may be due to cached file stats that return zero for the size of the file. To fix this issue, call `clearstatcache(false, $file)` with the path to the binary file.

Creating a JSON Response

Any type of response can be created via the *Response*[73] class by setting the right content and headers. A JSON response might look like this:

Listing 64-25
```
1 use Symfony\Component\HttpFoundation\Response;
2
3 $response = new Response();
4 $response->setContent(json_encode(array(
5     'data' => 123,
6 )));
7 $response->headers->set('Content-Type', 'application/json');
```

There is also a helpful *JsonResponse*[74] class, which can make this even easier:

Listing 64-26
```
1 use Symfony\Component\HttpFoundation\JsonResponse;
2
3 $response = new JsonResponse();
4 $response->setData(array(
5     'data' => 123
6 ));
```

This encodes your array of data to JSON and sets the `Content-Type` header to `application/json`.

To avoid XSSI *JSON Hijacking*[75], you should pass an associative array as the outer-most array to `JsonResponse` and not an indexed array so that the final result is an object (e.g. `{"object": "not inside an array"}`) instead of an array (e.g. `[{"object": "inside an array"}]`). Read the *OWASP guidelines*[76] for more information.

Only methods that respond to GET requests are vulnerable to XSSI 'JSON Hijacking'. Methods responding to POST requests only remain unaffected.

72. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/BinaryFileResponse.html#method_deleteFileAfterSend
73. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html
74. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/JsonResponse.html
75. http://haacked.com/archive/2009/06/25/json-hijacking.aspx
76. https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines#Always_return_JSON_with_an_Object_on_the_outside

JSONP Callback

If you're using JSONP, you can set the callback function that the data should be passed to:

Listing 64-27
```
$response->setCallback('handleResponse');
```

In this case, the `Content-Type` header will be `text/javascript` and the response content will look like this:

Listing 64-28
```
1  handleResponse({'data': 123});
```

Session

The session information is in its own document: *Session Management*.

Chapter 65
Session Management

The Symfony HttpFoundation component has a very powerful and flexible session subsystem which is designed to provide session management through a simple object-oriented interface using a variety of session storage drivers.

Sessions are used via the simple `Session`[1] implementation of `SessionInterface`[2] interface.

Make sure your PHP session isn't already started before using the Session class. If you have a legacy session system that starts your session, see *Legacy Sessions*.

Quick example:

Listing 65-1

```
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
$session->start();

// set and get session attributes
$session->set('name', 'Drak');
$session->get('name');

// set flash messages
$session->getFlashBag()->add('notice', 'Profile updated');

// retrieve messages
foreach ($session->getFlashBag()->get('notice', array()) as $message) {
    echo '<div class="flash-notice">'.$message.'</div>';
}
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html
2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionInterface.html

Symfony sessions are designed to replace several native PHP functions. Applications should avoid using `session_start()`, `session_regenerate_id()`, `session_id()`, `session_name()`, and `session_destroy()` and instead use the APIs in the following section.

While it is recommended to explicitly start a session, a session will actually start on demand, that is, if any session request is made to read/write session data.

Symfony sessions are incompatible with `php.ini` directive `session.auto_start = 1` This directive should be turned off in `php.ini`, in the webserver directives or in `.htaccess`.

Session API

The *Session*[3] class implements *SessionInterface*[4].

The *Session*[5] has a simple API as follows divided into a couple of groups.

Session Workflow

start()[6]
: Starts the session - do not use `session_start()`.

migrate()[7]
: Regenerates the session ID - do not use `session_regenerate_id()`. This method can optionally change the lifetime of the new cookie that will be emitted by calling this method.

invalidate()[8]
: Clears all session data and regenerates session ID. Do not use `session_destroy()`.

getId()[9]
: Gets the session ID. Do not use `session_id()`.

setId()[10]
: Sets the session ID. Do not use `session_id()`.

getName()[11]
: Gets the session name. Do not use `session_name()`.

setName()[12]
: Sets the session name. Do not use `session_name()`.

3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html
4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionInterface.html
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html
6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_start
7. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_migrate
8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_invalidate
9. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getId
10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_setId
11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getName
12. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_setName

Session Attributes

set()[13]
: Sets an attribute by key.

get()[14]
: Gets an attribute by key.

all()[15]
: Gets all attributes as an array of key => value.

has()[16]
: Returns true if the attribute exists.

replace()[17]
: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.

remove()[18]
: Deletes an attribute by key.

clear()[19]
: Clear all attributes.

The attributes are stored internally in a "Bag", a PHP object that acts like an array. A few methods exist for "Bag" management:

registerBag()[20]
: Registers a *SessionBagInterface*[21].

getBag()[22]
: Gets a *SessionBagInterface*[23] by bag name.

getFlashBag()[24]
: Gets the *FlashBagInterface*[25]. This is just a shortcut for convenience.

Session Metadata

getMetadataBag()[26]
: Gets the *MetadataBag*[27] which contains information about the session.

13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_set
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_get
15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_all
16. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_has
17. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_replace
18. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_remove
19. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_clear
20. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_registerBag
21. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
22. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getBag
23. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
24. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getFlashBag
25. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
26. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getMetadataBag
27. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/MetadataBag.html

Session Data Management

PHP's session management requires the use of the `$_SESSION` super-global, however, this interferes somewhat with code testability and encapsulation in an OOP paradigm. To help overcome this, Symfony uses *session bags* linked to the session to encapsulate a specific dataset of attributes or flash messages.

This approach also mitigates namespace pollution within the `$_SESSION` super-global because each bag stores all its data under a unique namespace. This allows Symfony to peacefully co-exist with other applications or libraries that might use the `$_SESSION` super-global and all data remains completely compatible with Symfony's session management.

Symfony provides two kinds of storage bags, with two separate implementations. Everything is written against interfaces so you may extend or create your own bag types if necessary.

`SessionBagInterface`[28] has the following API which is intended mainly for internal purposes:

`getStorageKey()`[29]
: Returns the key which the bag will ultimately store its array under in `$_SESSION`. Generally this value can be left at its default and is for internal use.

`initialize()`[30]
: This is called internally by Symfony session storage classes to link bag data to the session.

`getName()`[31]
: Returns the name of the session bag.

Attributes

The purpose of the bags implementing the *`AttributeBagInterface`*[32] is to handle session attribute storage. This might include things like user ID, and remember me login settings or other user based state information.

`AttributeBag`[33]
: This is the standard default implementation.

`NamespacedAttributeBag`[34]
: This implementation allows for attributes to be stored in a structured namespace.

Any plain key-value storage system is limited in the extent to which complex data can be stored since each key must be unique. You can achieve namespacing by introducing a naming convention to the keys so different parts of your application could operate without clashing. For example, `module1.foo` and `module2.foo`. However, sometimes this is not very practical when the attributes data is an array, for example a set of tokens. In this case, managing the array becomes a burden because you have to retrieve the array then process it and store it again:

Listing 65-2
```
1 $tokens = array(
2     'tokens' => array(
3         'a' => 'a6c1e0b6',
4         'b' => 'f4a7b1f3',
```

28. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
29. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#method_getStorageKey
30. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#method_initialize
31. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#method_getName
32. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
33. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBag.html
34. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/NamespacedAttributeBag.html

```
5       ),
6   );
```

So any processing of this might quickly get ugly, even simply adding a token to the array:

Listing 65-3
```
$tokens = $session->get('tokens');
$tokens['c'] = $value;
$session->set('tokens', $tokens);
```

With structured namespacing, the key can be translated to the array structure like this using a namespace character (defaults to `/`):

Listing 65-4
```
$session->set('tokens/c', $value);
```

This way you can easily access a key within the stored array directly and easily.

`AttributeBagInterface`[35] has a simple API

`set()`[36]
: Sets an attribute by key.

`get()`[37]
: Gets an attribute by key.

`all()`[38]
: Gets all attributes as an array of key => value.

`has()`[39]
: Returns true if the attribute exists.

`replace()`[40]
: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.

`remove()`[41]
: Deletes an attribute by key.

`clear()`[42]
: Clear the bag.

Flash Messages

The purpose of the *`FlashBagInterface`*[43] is to provide a way of setting and retrieving messages on a per session basis. The usual workflow would be to set flash messages in a request and to display them after a page redirect. For example, a user submits a form which hits an update controller, and after processing the controller redirects the page to either the updated page or an error page. Flash messages set in the previous page request would be displayed immediately on the subsequent page load for that session. This is however just one application for flash messages.

35. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
36. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_set
37. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_get
38. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_all
39. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_has
40. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_replace
41. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_remove
42. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#method_clear
43. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html

AutoExpireFlashBag[44]
: In this implementation, messages set in one page-load will be available for display only on the next page load. These messages will auto expire regardless of if they are retrieved or not.

FlashBag[45]
: In this implementation, messages will remain in the session until they are explicitly retrieved or cleared. This makes it possible to use ESI caching.

FlashBagInterface[46] has a simple API

add()[47]
: Adds a flash message to the stack of specified type.

set()[48]
: Sets flashes by type; This method conveniently takes both single messages as a `string` or multiple messages in an `array`.

get()[49]
: Gets flashes by type and clears those flashes from the bag.

setAll()[50]
: Sets all flashes, accepts a keyed array of arrays `type => array(messages)`.

all()[51]
: Gets all flashes (as a keyed array of arrays) and clears the flashes from the bag.

peek()[52]
: Gets flashes by type (read only).

peekAll()[53]
: Gets all flashes (read only) as keyed array of arrays.

has()[54]
: Returns true if the type exists, false if not.

keys()[55]
: Returns an array of the stored flash types.

clear()[56]
: Clears the bag.

For simple applications it is usually sufficient to have one flash message per type, for example a confirmation notice after a form is submitted. However, flash messages are stored in a keyed array by flash `$type` which means your application can issue multiple messages for a given type. This allows the API to be used for more complex messaging in your application.

44. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/AutoExpireFlashBag.html
45. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBag.html
46. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
47. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_add
48. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_set
49. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_get
50. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_setAll
51. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_all
52. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_peek
53. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_peekAll
54. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_has
55. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_keys
56. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_clear

Examples of setting multiple flashes:

Listing 65-5

```
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
$session->start();

// add flash messages
$session->getFlashBag()->add(
    'warning',
    'Your config file is writable, it should be set read-only'
);
$session->getFlashBag()->add('error', 'Failed to update name');
$session->getFlashBag()->add('error', 'Another error');
```

Displaying the flash messages might look as follows.

Simple, display one type of message:

Listing 65-6

```
// display warnings
foreach ($session->getFlashBag()->get('warning', array()) as $message) {
    echo '<div class="flash-warning">'.$message.'</div>';
}

// display errors
foreach ($session->getFlashBag()->get('error', array()) as $message) {
    echo '<div class="flash-error">'.$message.'</div>';
}
```

Compact method to process display all flashes at once:

Listing 65-7

```
foreach ($session->getFlashBag()->all() as $type => $messages) {
    foreach ($messages as $message) {
        echo '<div class="flash-'.$type.'">'.$message.'</div>';
    }
}
```

Chapter 66
Configuring Sessions and Save Handlers

This section deals with how to configure session management and fine tune it to your specific needs. This documentation covers save handlers, which store and retrieve session data, and configuring session behavior.

Save Handlers

The PHP session workflow has 6 possible operations that may occur. The normal session follows `open`, `read`, `write` and `close`, with the possibility of `destroy` and `gc` (garbage collection which will expire any old sessions: `gc` is called randomly according to PHP's configuration and if called, it is invoked after the `open` operation). You can read more about this at *php.net/session.customhandler*[1]

Native PHP Save Handlers

So-called native handlers, are save handlers which are either compiled into PHP or provided by PHP extensions, such as PHP-Sqlite, PHP-Memcached and so on.

All native save handlers are internal to PHP and as such, have no public facing API. They must be configured by `php.ini` directives, usually `session.save_path` and potentially other driver specific directives. Specific details can be found in the docblock of the `setOptions()` method of each class. For instance, the one provided by the Memcached extension can be found on *php.net/memcached.setoption*[2]

While native save handlers can be activated by directly using `ini_set('session.save_handler', $name);`, Symfony provides a convenient way to activate these in the same way as it does for custom handlers.

Symfony provides drivers for the following native save handler as an example:

- *NativeFileSessionHandler*[3]

Example usage:

1. http://php.net/session.customhandler
2. http://php.net/memcached.setoption
3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeFileSessionHandler.html

Listing 66-1

```
1 use Symfony\Component\HttpFoundation\Session\Session;
2 use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
3 use Symfony\Component\HttpFoundation\Session\Storage\Handler\NativeFileSessionHandler;
4
5 $storage = new NativeSessionStorage(array(), new NativeFileSessionHandler());
6 $session = new Session($storage);
```

With the exception of the `files` handler which is built into PHP and always available, the availability of the other handlers depends on those PHP extensions being active at runtime.

Native save handlers provide a quick solution to session storage, however, in complex systems where you need more control, custom save handlers may provide more freedom and flexibility. Symfony provides several implementations which you may further customize as required.

Custom Save Handlers

Custom handlers are those which completely replace PHP's built-in session save handlers by providing six callback functions which PHP calls internally at various points in the session workflow.

The Symfony HttpFoundation component provides some by default and these can easily serve as examples if you wish to write your own.

- *PdoSessionHandler*[4]
- *MemcacheSessionHandler*[5]
- *MemcachedSessionHandler*[6]
- *MongoDbSessionHandler*[7]
- *NullSessionHandler*[8]

Example usage:

Listing 66-2

```
1 use Symfony\Component\HttpFoundation\Session\Session;
2 use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
3 use Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler;
4
5 $pdo = new \PDO(...);
6 $storage = new NativeSessionStorage(array(), new PdoSessionHandler($pdo));
7 $session = new Session($storage);
```

Configuring PHP Sessions

The *NativeSessionStorage*[9] can configure most of the `php.ini` configuration directives which are documented at *php.net/session.configuration*[10].

4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcacheSessionHandler.html
6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcachedSessionHandler.html
7. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/MongoDbSessionHandler.html
8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/NullSessionHandler.html
9. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
10. http://php.net/session.configuration

To configure these settings, pass the keys (omitting the initial `session.` part of the key) as a key-value array to the `$options` constructor argument. Or set them via the *setOptions()*[11] method.

For the sake of clarity, some key options are explained in this documentation.

Session Cookie Lifetime

For security, session tokens are generally recommended to be sent as session cookies. You can configure the lifetime of session cookies by specifying the lifetime (in seconds) using the `cookie_lifetime` key in the constructor's `$options` argument in *NativeSessionStorage*[12].

Setting a `cookie_lifetime` to `0` will cause the cookie to live only as long as the browser remains open. Generally, `cookie_lifetime` would be set to a relatively large number of days, weeks or months. It is not uncommon to set cookies for a year or more depending on the application.

Since session cookies are just a client-side token, they are less important in controlling the fine details of your security settings which ultimately can only be securely controlled from the server side.

The `cookie_lifetime` setting is the number of seconds the cookie should live for, it is not a Unix timestamp. The resulting session cookie will be stamped with an expiry time of `time()` + `cookie_lifetime` where the time is taken from the server.

Configuring Garbage Collection

When a session opens, PHP will call the `gc` handler randomly according to the probability set by `session.gc_probability` / `session.gc_divisor`. For example if these were set to `5/100` respectively, it would mean a probability of 5%. Similarly, `3/4` would mean a 3 in 4 chance of being called, i.e. 75%.

If the garbage collection handler is invoked, PHP will pass the value stored in the `php.ini` directive `session.gc_maxlifetime`. The meaning in this context is that any stored session that was saved more than `gc_maxlifetime` ago should be deleted. This allows one to expire records based on idle time.

You can configure these settings by passing `gc_probability`, `gc_divisor` and `gc_maxlifetime` in an array to the constructor of *NativeSessionStorage*[13] or to the *setOptions()*[14] method.

Session Lifetime

When a new session is created, meaning Symfony issues a new session cookie to the client, the cookie will be stamped with an expiry time. This is calculated by adding the PHP runtime configuration value in `session.cookie_lifetime` with the current server time.

PHP will only issue a cookie once. The client is expected to store that cookie for the entire lifetime. A new cookie will only be issued when the session is destroyed, the browser cookie is deleted, or the session ID is regenerated using the `migrate()` or `invalidate()` methods of the `Session` class.

The initial cookie lifetime can be set by configuring `NativeSessionStorage` using the `setOptions(array('cookie_lifetime' => 1234))` method.

11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#method_setOptions
12. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#method_setOptions

A cookie lifetime of `0` means the cookie expires when the browser is closed.

Session Idle Time/Keep Alive

There are often circumstances where you may want to protect, or minimize unauthorized use of a session when a user steps away from their terminal while logged in by destroying the session after a certain period of idle time. For example, it is common for banking applications to log the user out after just 5 to 10 minutes of inactivity. Setting the cookie lifetime here is not appropriate because that can be manipulated by the client, so we must do the expiry on the server side. The easiest way is to implement this via garbage collection which runs reasonably frequently. The `cookie_lifetime` would be set to a relatively high value, and the garbage collection `gc_maxlifetime` would be set to destroy sessions at whatever the desired idle period is.

The other option is specifically check if a session has expired after the session is started. The session can be destroyed as required. This method of processing can allow the expiry of sessions to be integrated into the user experience, for example, by displaying a message.

Symfony records some basic metadata about each session to give you complete freedom in this area.

Session Cache Limiting

To avoid users seeing stale data, it's common for session-enabled resources to be sent with headers that disable caching. For this purpose PHP Sessions has the `sessions.cache_limiter` option, which determines which headers, if any, will be sent with the response when the session in started.

Upon construction, *`NativeSessionStorage`*[15] sets this global option to `""` (send no headers) in case the developer wishes to use a *`Response`*[16] object to manage response headers.

If you rely on PHP Sessions to manage HTTP caching, you *must* manually set the `cache_limiter` option in *`NativeSessionStorage`*[17] to a non-empty value.

For example, you may set it to PHP's default value during construction:

Example usage:

Listing 66-3

```
use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;

$options['cache_limiter'] = session_cache_limiter();
$storage = new NativeSessionStorage($options);
```

Session Metadata

Sessions are decorated with some basic metadata to enable fine control over the security settings. The session object has a getter for the metadata, *`getMetadataBag()`*[18] which exposes an instance of *`MetadataBag`*[19]:

15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
16. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html
17. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
18. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html#method_getMetadataBag

Listing 66-4
```
$session->getMetadataBag()->getCreated();
$session->getMetadataBag()->getLastUsed();
```

Both methods return a Unix timestamp (relative to the server).

This metadata can be used to explicitly expire a session on access, e.g.:

Listing 66-5
```
1 $session->start();
2 if (time() - $session->getMetadataBag()->getLastUsed() > $maxIdleTime) {
3     $session->invalidate();
4     throw new SessionExpired(); // redirect to expired session page
5 }
```

It is also possible to tell what the `cookie_lifetime` was set to for a particular cookie by reading the `getLifetime()` method:

Listing 66-6
```
$session->getMetadataBag()->getLifetime();
```

The expiry time of the cookie can be determined by adding the created timestamp and the lifetime.

PHP 5.4 Compatibility

Since PHP 5.4.0, *SessionHandler*[20] and *SessionHandlerInterface*[21] are available. Symfony provides forward compatibility for the *SessionHandlerInterface*[22] so it can be used under PHP 5.3. This greatly improves interoperability with other libraries.

SessionHandler[23] is a special PHP internal class which exposes native save handlers to PHP user-space.

In order to provide a solution for those using PHP 5.4, Symfony has a special class called *NativeSessionHandler*[24] which under PHP 5.4, extends from `\SessionHandler` and under PHP 5.3 is just a empty base class. This provides some interesting opportunities to leverage PHP 5.4 functionality if it is available.

Save Handler Proxy

A Save Handler Proxy is basically a wrapper around a Save Handler that was introduced to seamlessly support the migration from PHP 5.3 to PHP 5.4+. It further creates an extension point from where custom logic can be added that works independently of which handler is being wrapped inside.

There are two kinds of save handler class proxies which inherit from *AbstractProxy*[25]: they are *NativeProxy*[26] and *SessionHandlerProxy*[27].

NativeSessionStorage[28] automatically injects storage handlers into a save handler proxy unless already wrapped by one.

19. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/MetadataBag.html
20. http://php.net/manual/en/class.sessionhandler.php
21. http://php.net/manual/en/class.sessionhandlerinterface.php
22. http://php.net/manual/en/class.sessionhandlerinterface.php
23. http://php.net/manual/en/class.sessionhandler.php
24. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeSessionHandler.html
25. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/AbstractProxy.html
26. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/NativeProxy.html
27. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/SessionHandlerProxy.html
28. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html

NativeProxy[29] is used automatically under PHP 5.3 when internal PHP save handlers are specified using the `Native*SessionHandler` classes, while *SessionHandlerProxy*[30] will be used to wrap any custom save handlers, that implement *SessionHandlerInterface*[31].

From PHP 5.4 and above, all session handlers implement *SessionHandlerInterface*[32] including `Native*SessionHandler` classes which inherit from *SessionHandler*[33].

The proxy mechanism allows you to get more deeply involved in session save handler classes. A proxy for example could be used to encrypt any session transaction without knowledge of the specific save handler.

Before PHP 5.4, you can only proxy user-land save handlers but not native PHP save handlers.

29. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/NativeProxy.html
30. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/SessionHandlerProxy.html
31. http://php.net/manual/en/class.sessionhandlerinterface.php
32. http://php.net/manual/en/class.sessionhandlerinterface.php
33. http://php.net/manual/en/class.sessionhandler.php

Chapter 67
Testing with Sessions

Symfony is designed from the ground up with code-testability in mind. In order to make your code which utilizes session easily testable we provide two separate mock storage mechanisms for both unit testing and functional testing.

Testing code using real sessions is tricky because PHP's workflow state is global and it is not possible to have multiple concurrent sessions in the same PHP process.

The mock storage engines simulate the PHP session workflow without actually starting one allowing you to test your code without complications. You may also run multiple instances in the same PHP process.

The mock storage drivers do not read or write the system globals `session_id()` or `session_name()`. Methods are provided to simulate this if required:

- *getId()*[1]: Gets the session ID.
- *setId()*[2]: Sets the session ID.
- *getName()*[3]: Gets the session name.
- *setName()*[4]: Sets the session name.

Unit Testing

For unit testing where it is not necessary to persist the session, you should simply swap out the default storage engine with *MockArraySessionStorage*[5]:

Listing 67-1
```
use Symfony\Component\HttpFoundation\Session\Storage\MockArraySessionStorage;
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session(new MockArraySessionStorage());
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.html#method_getId
2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.html#method_setId
3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.html#method_getName
4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/SessionStorageInterface.html#method_setName
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/MockArraySessionStorage.html

Functional Testing

For functional testing where you may need to persist session data across separate PHP processes, simply change the storage engine to *MockFileSessionStorage*[6]:

Listing 67-2
```
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\MockFileSessionStorage;

$session = new Session(new MockFileSessionStorage());
```

6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/MockFileSessionStorage.html

Chapter 68

Integrating with Legacy Sessions

Sometimes it may be necessary to integrate Symfony into a legacy application where you do not initially have the level of control you require.

As stated elsewhere, Symfony Sessions are designed to replace the use of PHP's native `session_*()` functions and use of the `$_SESSION` superglobal. Additionally, it is mandatory for Symfony to start the session.

However when there really are circumstances where this is not possible, you can use a special storage bridge *PhpBridgeSessionStorage*[1] which is designed to allow Symfony to work with a session started outside of the Symfony HttpFoundation component. You are warned that things can interrupt this use-case unless you are careful: for example the legacy application erases `$_SESSION`.

A typical use of this might look like this:

Listing 68-1

```
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\PhpBridgeSessionStorage;

// legacy application configures session
ini_set('session.save_handler', 'files');
ini_set('session.save_path', '/tmp');
session_start();

// Get Symfony to interface with this existing session
$session = new Session(new PhpBridgeSessionStorage());

// symfony will now interface with the existing PHP session
$session->start();
```

This will allow you to start using the Symfony Session API and allow migration of your application to Symfony sessions.

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/PhpBridgeSessionStorage.html

Symfony sessions store data like attributes in special 'Bags' which use a key in the `$_SESSION` superglobal. This means that a Symfony session cannot access arbitrary keys in `$_SESSION` that may be set by the legacy application, although all the `$_SESSION` contents will be saved when the session is saved.

Chapter 69

Trusting Proxies

If you're using the Symfony Framework, start by reading *How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy*.

If you find yourself behind some sort of proxy - like a load balancer - then certain header information may be sent to you using special `X-Forwarded-*` headers. For example, the `Host` HTTP header is usually used to return the requested host. But when you're behind a proxy, the true host may be stored in a `X-Forwarded-Host` header.

Since HTTP headers can be spoofed, Symfony does *not* trust these proxy headers by default. If you are behind a proxy, you should manually whitelist your proxy.

Listing 69-1

```
1  use Symfony\Component\HttpFoundation\Request;
2
3  // only trust proxy headers coming from this IP addresses
4  Request::setTrustedProxies(array('192.0.0.1', '10.0.0.0/8'));
```

Configuring Header Names

By default, the following proxy headers are trusted:

- `X-Forwarded-For` Used in *`getClientIp()`*[1];
- `X-Forwarded-Host` Used in *`getHost()`*[2];
- `X-Forwarded-Port` Used in *`getPort()`*[3];
- `X-Forwarded-Proto` Used in *`getScheme()`*[4] and *`isSecure()`*[5];

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getClientIp
2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getHost
3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getPort
4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getScheme
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_isSecure

If your reverse proxy uses a different header name for any of these, you can configure that header name via *setTrustedHeaderName()*[6]:

Listing 69-2
```
Request::setTrustedHeaderName(Request::HEADER_CLIENT_IP, 'X-Proxy-For');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_HOST, 'X-Proxy-Host');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PORT, 'X-Proxy-Port');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PROTO, 'X-Proxy-Proto');
```

Not Trusting certain Headers

By default, if you whitelist your proxy's IP address, then all four headers listed above are trusted. If you need to trust some of these headers but not others, you can do that as well:

Listing 69-3
```
// disables trusting the ``X-Forwarded-Proto`` header, the default header is used
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PROTO, '');
```

6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_setTrustedHeaderName

Chapter 70

The HttpKernel Component

The HttpKernel component provides a structured process for converting a `Request` into a `Response` by making use of the EventDispatcher component. It's flexible enough to create a full-stack framework (Symfony), a micro-framework (Silex) or an advanced CMS system (Drupal).

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/http-kernel` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/http-kernel*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

The Workflow of a Request

Every HTTP web interaction begins with a request and ends with a response. Your job as a developer is to create PHP code that reads the request information (e.g. the URL) and creates and returns a response (e.g. an HTML page or JSON string).

1. `https://packagist.org/packages/symfony/http-kernel`

The web in action

The User asks for a Resource in a Browser

The Browser sends a Request to the Server

Symfony gives the Developer a Request Object

The Developer "converts" the Request Object to a Response Object

The Server sends back a Response to the Browser

The Browser displays the Resource to the User

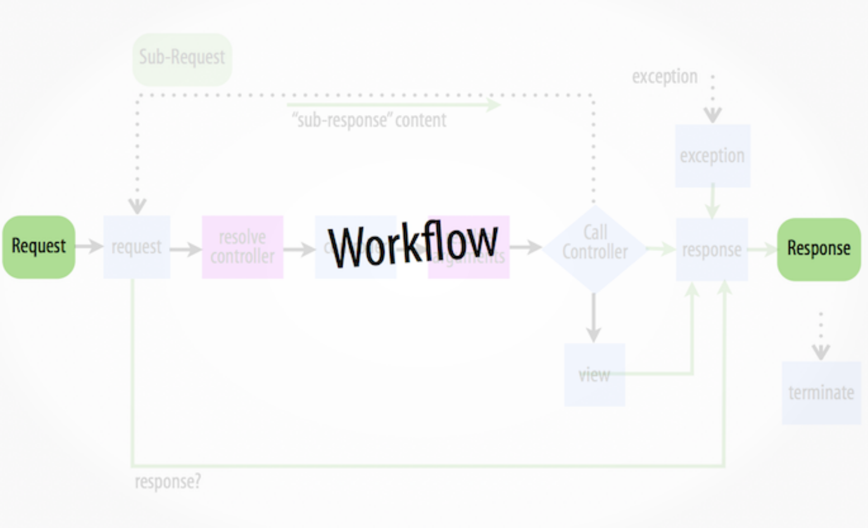
Typically, some sort of framework or system is built to handle all the repetitive tasks (e.g. routing, security, etc) so that a developer can easily build each *page* of the application. Exactly *how* these systems are built varies greatly. The HttpKernel component provides an interface that formalizes the process of starting with a request and creating the appropriate response. The component is meant to be the heart of any application or framework, no matter how varied the architecture of that system:

Listing 70-1

```
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;

interface HttpKernelInterface
{
    // ...

    /**
     * @return Response A Response instance
     */
    public function handle(
        Request $request,
        $type = self::MASTER_REQUEST,
        $catch = true
    );
}
```

Internally, `HttpKernel::handle()`[2] - the concrete implementation of `HttpKernelInterface::handle()`[3] - defines a workflow that starts with a `Request`[4] and ends with a `Response`[5].

2. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html#method_handle
3. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernelInterface.html#method_handle
4. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html



The exact details of this workflow are the key to understanding how the kernel (and the Symfony Framework or any other library that uses the kernel) works.

HttpKernel: Driven by Events

The `HttpKernel::handle()` method works internally by dispatching events. This makes the method both flexible, but also a bit abstract, since all the "work" of a framework/application built with HttpKernel is actually done in event listeners.

To help explain this process, this document looks at each step of the process and talks about how one specific implementation of the HttpKernel - the Symfony Framework - works.

Initially, using the *`HttpKernel`*[6] is really simple and involves creating an *event dispatcher* and a controller resolver (explained below). To complete your working kernel, you'll add more event listeners to the events discussed below:

Listing 70-2

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

// create the Request object
$request = Request::createFromGlobals();

$dispatcher = new EventDispatcher();
// ... add some event listeners

// create your controller resolver
$resolver = new ControllerResolver();
// instantiate the kernel
$kernel = new HttpKernel($dispatcher, $resolver);

// actually execute the kernel, which turns the request into a response
// by dispatching events, calling a controller, and returning the response
$response = $kernel->handle($request);

// send the headers and echo the content
$response->send();
```

6. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html

```
23
24 // triggers the kernel.terminate event
25 $kernel->terminate($request, $response);
```

See "A full Working Example" for a more concrete implementation.

For general information on adding listeners to the events below, see Creating an Event Listener.

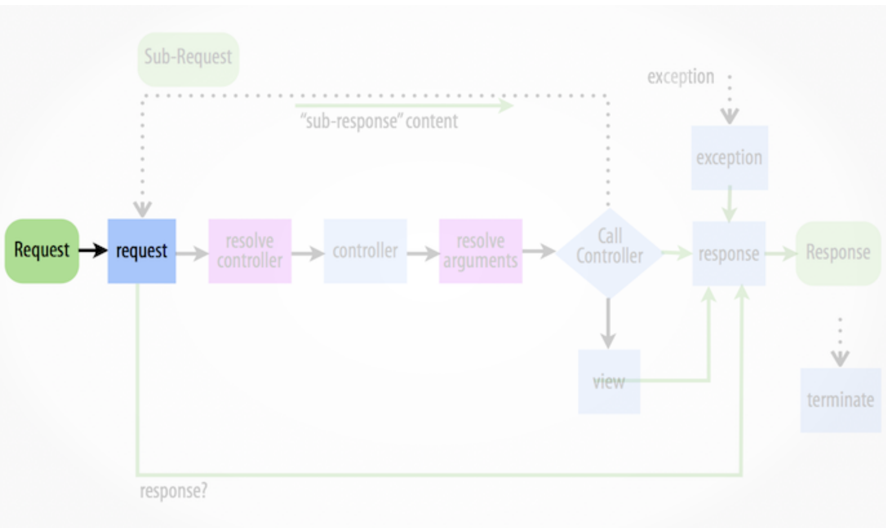
> *There is a wonderful tutorial series on using the HttpKernel component and other Symfony components to create your own framework. See Introduction.*

1) The `kernel.request` Event

Typical Purposes: To add more information to the `Request`, initialize parts of the system, or return a `Response` if possible (e.g. a security layer that denies access).

Kernel Events Information Table

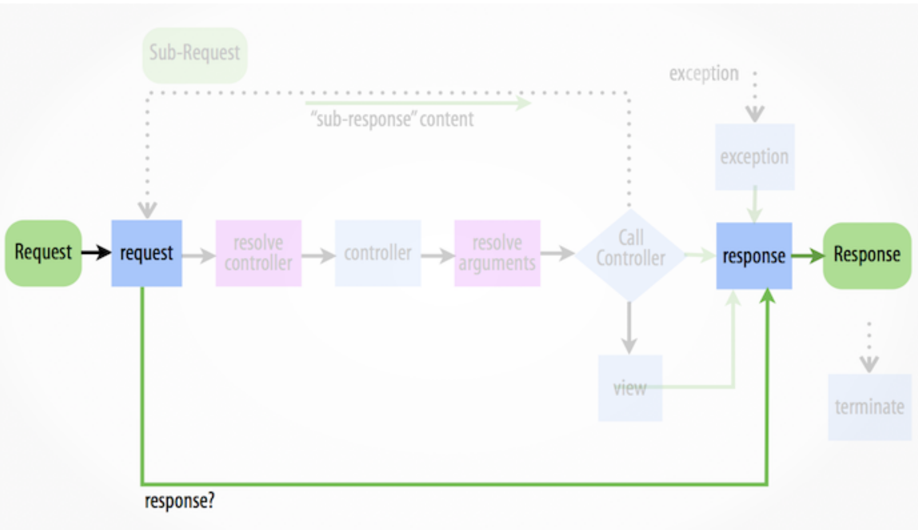
The first event that is dispatched inside *`HttpKernel::handle`*[7] is `kernel.request`, which may have a variety of different listeners.



Listeners of this event can be quite varied. Some listeners - such as a security listener - might have enough information to create a `Response` object immediately. For example, if a security listener determined that a user doesn't have access, that listener may return a *`RedirectResponse`*[8] to the login page or a 403 Access Denied response.

If a `Response` is returned at this stage, the process skips directly to the kernel.response event.

7. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html#method_handle
8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RedirectResponse.html



Other listeners simply initialize things or add more information to the request. For example, a listener might determine and set the locale on the `Request` object.

Another common listener is routing. A router listener may process the `Request` and determine the controller that should be rendered (see the next section). In fact, the `Request` object has an "attributes" bag which is a perfect spot to store this extra, application-specific data about the request. This means that if your router listener somehow determines the controller, it can store it on the `Request` attributes (which can be used by your controller resolver).

Overall, the purpose of the `kernel.request` event is either to create and return a `Response` directly, or to add information to the `Request` (e.g. setting the locale or setting some other information on the `Request` attributes).

When setting a response for the `kernel.request` event, the propagation is stopped. This means listeners with lower priority won't be executed.

`kernel.request` in the Symfony Framework

The most important listener to `kernel.request` in the Symfony Framework is the *`RouterListener`*[9]. This class executes the routing layer, which returns an *array* of information about the matched request, including the `_controller` and any placeholders that are in the route's pattern (e.g. `{slug}`). See *Routing component*.

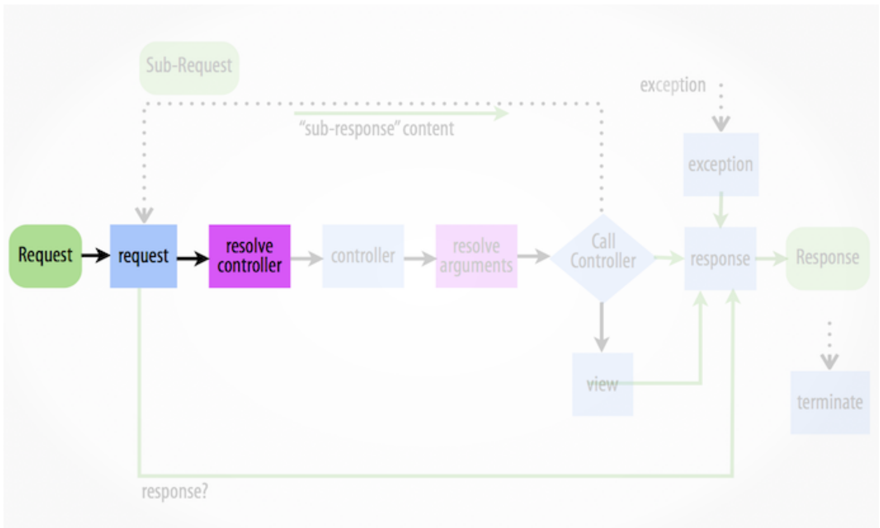
This array of information is stored in the *`Request`*[10] object's `attributes` array. Adding the routing information here doesn't do anything yet, but is used next when resolving the controller.

2) Resolve the Controller

Assuming that no `kernel.request` listener was able to create a `Response`, the next step in HttpKernel is to determine and prepare (i.e. resolve) the controller. The controller is the part of the end-application's code that is responsible for creating and returning the `Response` for a specific page. The only requirement is that it is a PHP callable - i.e. a function, method on an object, or a `Closure`.

9. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/EventListener/RouterListener.html
10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html

But *how* you determine the exact controller for a request is entirely up to your application. This is the job of the "controller resolver" - a class that implements *ControllerResolverInterface*[11] and is one of the constructor arguments to `HttpKernel`.



Your job is to create a class that implements the interface and fill in its two methods: `getController` and `getArguments`. In fact, one default implementation already exists, which you can use directly or learn from: *ControllerResolver*[12]. This implementation is explained more in the sidebar below:

Listing 70-3

```
namespace Symfony\Component\HttpKernel\Controller;

use Symfony\Component\HttpFoundation\Request;

interface ControllerResolverInterface
{
    public function getController(Request $request);

    public function getArguments(Request $request, $controller);
}
```

Internally, the `HttpKernel::handle` method first calls *getController()*[13] on the controller resolver. This method is passed the `Request` and is responsible for somehow determining and returning a PHP callable (the controller) based on the request's information.

The second method, *getArguments()*[14], will be called after another event - `kernel.controller` - is dispatched.

11. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html
12. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
13. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#method_getController
14. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#method_getArguments

Resolving the Controller in the Symfony Framework

The Symfony Framework uses the built-in *ControllerResolver*[15] class (actually, it uses a sub-class with some extra functionality mentioned below). This class leverages the information that was placed on the `Request` object's `attributes` property during the `RouterListener`.

getController

The `ControllerResolver` looks for a `_controller` key on the `Request` object's attributes property (recall that this information is typically placed on the `Request` via the `RouterListener`). This string is then transformed into a PHP callable by doing the following:

1. The `AcmeDemoBundle:Default:index` format of the `_controller` key is changed to another string that contains the full class and method name of the controller by following the convention used in Symfony - e.g. `Acme\DemoBundle\Controller\DefaultController::indexAction`. This transformation is specific to the *ControllerResolver*[16] sub-class used by the Symfony Framework.
2. A new instance of your controller class is instantiated with no constructor arguments.
3. If the controller implements *ContainerAwareInterface*[17], `setContainer` is called on the controller object and the container is passed to it. This step is also specific to the *ControllerResolver*[18] sub-class used by the Symfony Framework.

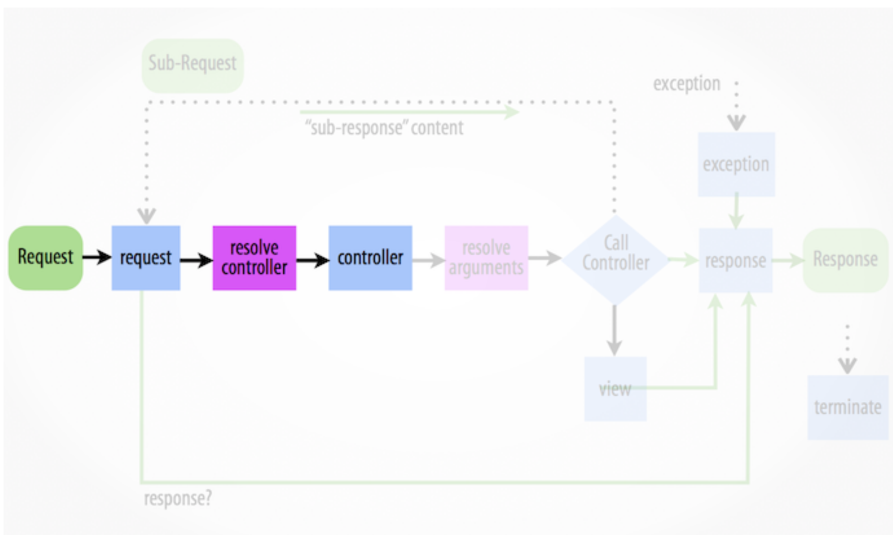
   There are also a few other variations on the above process (e.g. if you're registering your controllers as services).

3) The `kernel.controller` Event

Typical Purposes: Initialize things or change the controller just before the controller is executed.

Kernel Events Information Table

After the controller callable has been determined, `HttpKernel::handle` dispatches the `kernel.controller` event. Listeners to this event might initialize some part of the system that needs to be initialized after certain things have been determined (e.g. the controller, routing information) but before the controller is executed. For some examples, see the Symfony section below.



15. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
16. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html
17. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerAwareInterface.html
18. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html

Listeners to this event can also change the controller callable completely by calling *`FilterControllerEvent::setController`*[19] on the event object that's passed to listeners on this event.

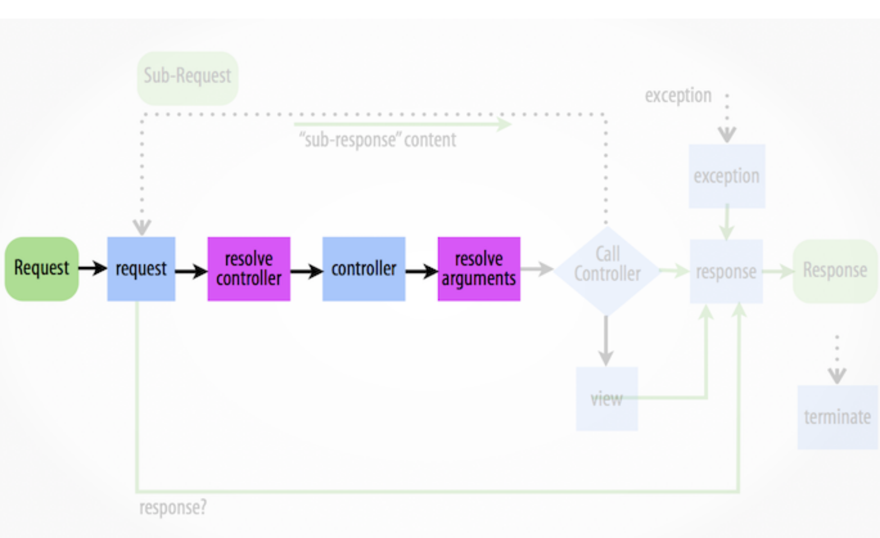
`kernel.controller` in the Symfony Framework

There are a few minor listeners to the `kernel.controller` event in the Symfony Framework, and many deal with collecting profiler data when the profiler is enabled.

One interesting listener comes from the *SensioFrameworkExtraBundle*[20], which is packaged with the Symfony Standard Edition. This listener's *@ParamConverter*[21] functionality allows you to pass a full object (e.g. a `Post` object) to your controller instead of a scalar value (e.g. an `id` parameter that was on your route). The listener - `ParamConverterListener` - uses reflection to look at each of the arguments of the controller and tries to use different methods to convert those to objects, which are then stored in the `attributes` property of the `Request` object. Read the next section to see why this is important.

4) Getting the Controller Arguments

Next, `HttpKernel::handle` calls *`getArguments()`*[22]. Remember that the controller returned in `getController` is a callable. The purpose of `getArguments` is to return the array of arguments that should be passed to that controller. Exactly how this is done is completely up to your design, though the built-in *`ControllerResolver`*[23] is a good example.



At this point the kernel has a PHP callable (the controller) and an array of arguments that should be passed when executing that callable.

19. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html#method_setController
20. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html
21. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html
22. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#method_getArguments
23. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html

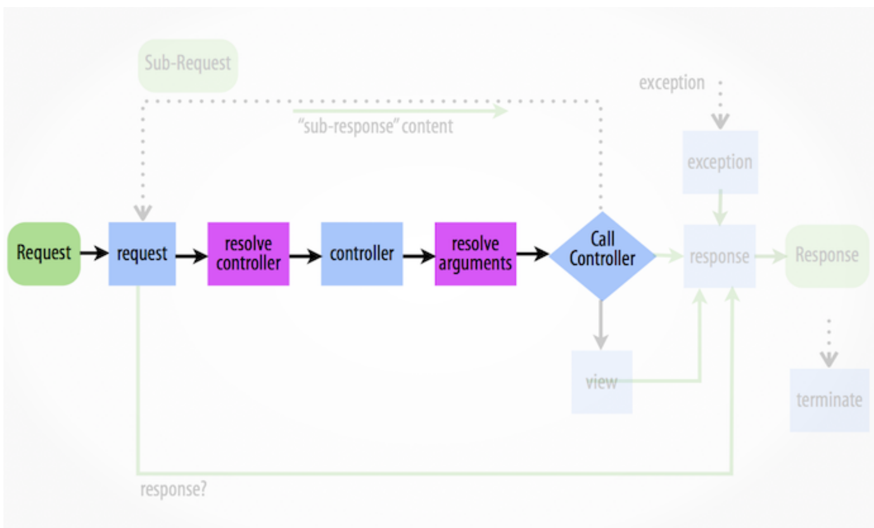
Getting the Controller Arguments in the Symfony Framework

Now that you know exactly what the controller callable (usually a method inside a controller object) is, the `ControllerResolver` uses *reflection*[24] on the callable to return an array of the *names* of each of the arguments. It then iterates over each of these arguments and uses the following tricks to determine which value should be passed for each argument:

1. If the `Request` attributes bag contains a key that matches the name of the argument, that value is used. For example, if the first argument to a controller is `$slug`, and there is a `slug` key in the `Request` `attributes` bag, that value is used (and typically this value came from the `RouterListener`).
2. If the argument in the controller is type-hinted with Symfony's *`Request`*[25] object, then the `Request` is passed in as the value.

5) Calling the Controller

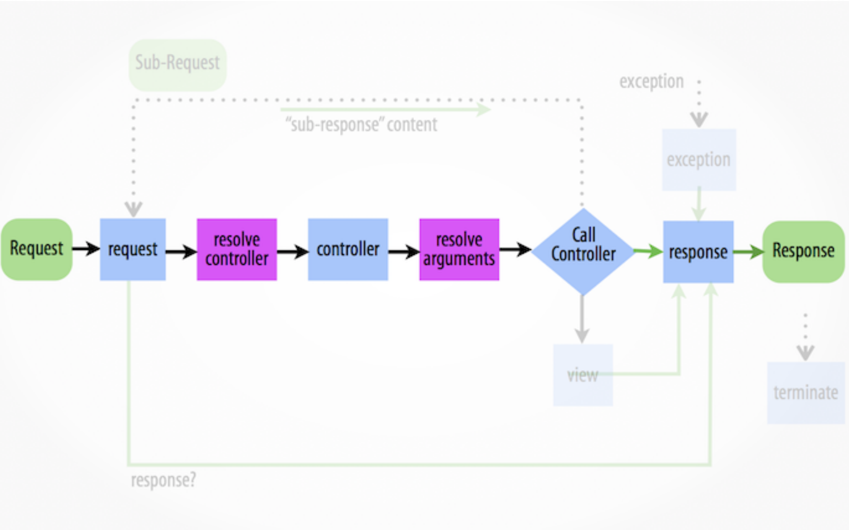
The next step is simple! `HttpKernel::handle` executes the controller.



The job of the controller is to build the response for the given resource. This could be an HTML page, a JSON string or anything else. Unlike every other part of the process so far, this step is implemented by the "end-developer", for each page that is built.

Usually, the controller will return a `Response` object. If this is true, then the work of the kernel is just about done! In this case, the next step is the kernel.response event.

24. `http://php.net/manual/en/book.reflection.php`
25. `http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html`



But if the controller returns anything besides a `Response`, then the kernel has a little bit more work to do - kernel.view (since the end goal is *always* to generate a `Response` object).

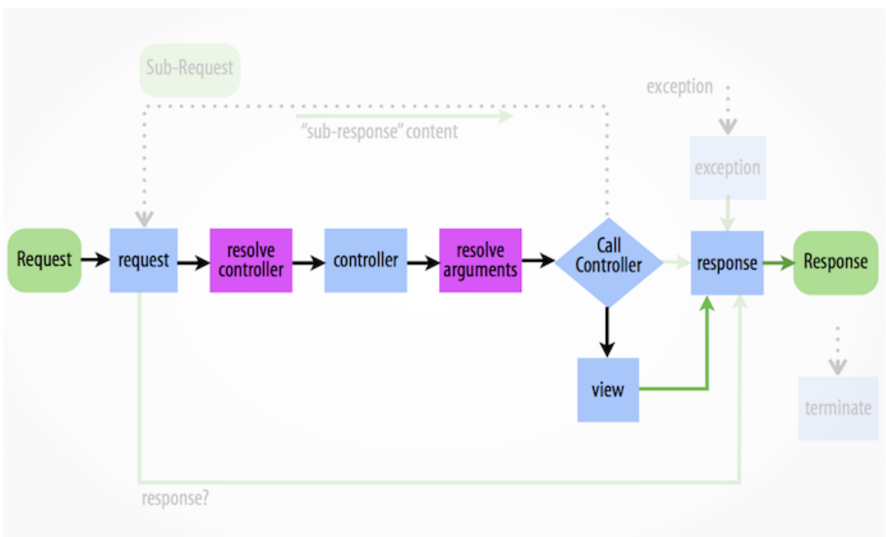
A controller must return *something*. If a controller returns `null`, an exception will be thrown immediately.

6) The `kernel.view` Event

Typical Purposes: Transform a non-`Response` return value from a controller into a `Response`

Kernel Events Information Table

If the controller doesn't return a `Response` object, then the kernel dispatches another event - `kernel.view`. The job of a listener to this event is to use the return value of the controller (e.g. an array of data or an object) to create a `Response`.



This can be useful if you want to use a "view" layer: instead of returning a `Response` from the controller, you return data that represents the page. A listener to this event could then use this data to create a `Response` that is in the correct format (e.g HTML, JSON, etc).

At this stage, if no listener sets a response on the event, then an exception is thrown: either the controller *or* one of the view listeners must always return a `Response`.

When setting a response for the `kernel.view` event, the propagation is stopped. This means listeners with lower priority won't be executed.

`kernel.view` in the Symfony Framework

There is no default listener inside the Symfony Framework for the `kernel.view` event. However, one core bundle - *SensioFrameworkExtraBundle*[26] - *does* add a listener to this event. If your controller returns an array, and you place the *@Template*[27] annotation above the controller, then this listener renders a template, passes the array you returned from your controller to that template, and creates a `Response` containing the returned content from that template.

Additionally, a popular community bundle *FOSRestBundle*[28] implements a listener on this event which aims to give you a robust view layer capable of using a single controller to return many different content-type responses (e.g. HTML, JSON, XML, etc).

7) The `kernel.response` Event

Typical Purposes: Modify the `Response` object just before it is sent

Kernel Events Information Table

The end goal of the kernel is to transform a `Request` into a `Response`. The `Response` might be created during the kernel.request event, returned from the controller, or returned by one of the listeners to the kernel.view event.

Regardless of who creates the `Response`, another event - `kernel.response` is dispatched directly afterwards. A typical listener to this event will modify the `Response` object in some way, such as modifying headers, adding cookies, or even changing the content of the `Response` itself (e.g. injecting some JavaScript before the end `</body>` tag of an HTML response).

After this event is dispatched, the final `Response` object is returned from *`handle()`*[29]. In the most typical use-case, you can then call the *`send()`*[30] method, which sends the headers and prints the `Response` content.

`kernel.response` in the Symfony Framework

There are several minor listeners on this event inside the Symfony Framework, and most modify the response in some way. For example, the *`WebDebugToolbarListener`*[31] injects some JavaScript at the bottom of your page in the `dev` environment which causes the web debug toolbar to be displayed. Another listener, *`ContextListener`*[32] serializes the current user's information into the session so that it can be reloaded on the next request.

26. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html
27. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/view.html
28. https://github.com/friendsofsymfony/FOSRestBundle
29. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html#method_handle
30. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_send
31. http://api.symfony.com/3.0/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
32. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/ContextListener.html

8) The `kernel.terminate` Event

Typical Purposes: To perform some "heavy" action after the response has been streamed to the user

Kernel Events Information Table

The final event of the HttpKernel process is `kernel.terminate` and is unique because it occurs *after* the `HttpKernel::handle` method, and after the response is sent to the user. Recall from above, then the code that uses the kernel, ends like this:

Listing 70-4

```
// send the headers and echo the content
$response->send();

// triggers the kernel.terminate event
$kernel->terminate($request, $response);
```

As you can see, by calling `$kernel->terminate` after sending the response, you will trigger the `kernel.terminate` event where you can perform certain actions that you may have delayed in order to return the response as quickly as possible to the client (e.g. sending emails).

Internally, the HttpKernel makes use of the *`fastcgi_finish_request`*[33] PHP function. This means that at the moment, only the *PHP FPM*[34] server API is able to send a response to the client while the server's PHP process still performs some tasks. With all other server APIs, listeners to `kernel.terminate` are still executed, but the response is not sent to the client until they are all completed.

Using the `kernel.terminate` event is optional, and should only be called if your kernel implements *`TerminableInterface`*[35].

`kernel.terminate` in the Symfony Framework

If you use the SwiftmailerBundle with Symfony and use `memory` spooling, then the *EmailSenderListener*[36] is activated, which actually delivers any emails that you scheduled to send during the request.

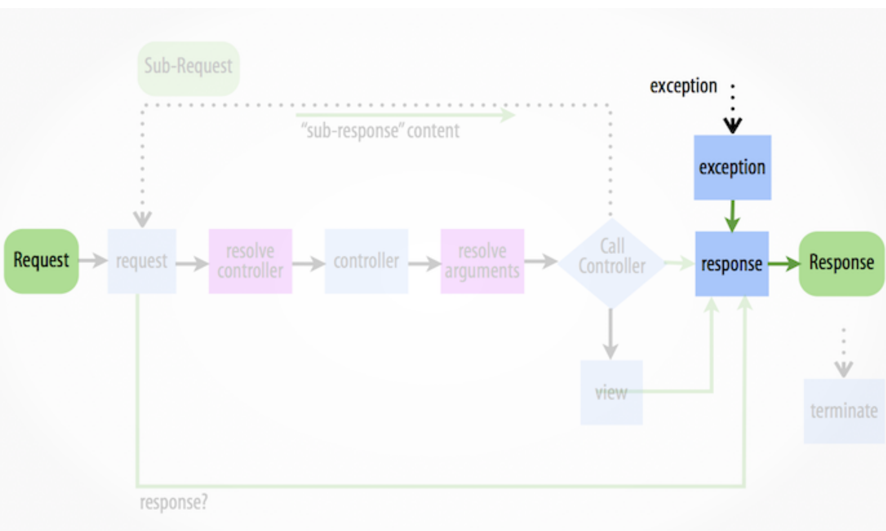
Handling Exceptions: the `kernel.exception` Event

Typical Purposes: Handle some type of exception and create an appropriate `Response` to return for the exception

Kernel Events Information Table

If an exception is thrown at any point inside `HttpKernel::handle`, another event - `kernel.exception` is thrown. Internally, the body of the `handle` function is wrapped in a try-catch block. When any exception is thrown, the `kernel.exception` event is dispatched so that your system can somehow respond to the exception.

33. http://php.net/manual/en/function.fastcgi-finish-request.php
34. http://php.net/manual/en/install.fpm.php
35. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/TerminableInterface.html
36. https://github.com/symfony/swiftmailer-bundle/blob/master/EventListener/EmailSenderListener.php



Each listener to this event is passed a *GetResponseForExceptionEvent*[37] object, which you can use to access the original exception via the *getException()*[38] method. A typical listener on this event will check for a certain type of exception and create an appropriate error `Response`.

For example, to generate a 404 page, you might throw a special type of exception and then add a listener on this event that looks for this exception and creates and returns a 404 `Response`. In fact, the HttpKernel component comes with an *ExceptionListener*[39], which if you choose to use, will do this and more by default (see the sidebar below for more details).

When setting a response for the `kernel.exception` event, the propagation is stopped. This means listeners with lower priority won't be executed.

37. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
38. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html#method_getException
39. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html

`kernel.exception` in the Symfony Framework

There are two main listeners to `kernel.exception` when using the Symfony Framework.

ExceptionListener in HttpKernel

The first comes core to the HttpKernel component and is called *ExceptionListener*[40]. The listener has several goals:

1. The thrown exception is converted into a *FlattenException*[41] object, which contains all the information about the request, but which can be printed and serialized.
2. If the original exception implements *HttpExceptionInterface*[42], then `getStatusCode` and `getHeaders` are called on the exception and used to populate the headers and status code of the `FlattenException` object. The idea is that these are used in the next step when creating the final response.
3. A controller is executed and passed the flattened exception. The exact controller to render is passed as a constructor argument to this listener. This controller will return the final `Response` for this error page.

ExceptionListener in Security

The other important listener is the *ExceptionListener*[43]. The goal of this listener is to handle security exceptions and, when appropriate, *help* the user to authenticate (e.g. redirect to the login page).

Creating an Event Listener

As you've seen, you can create and attach event listeners to any of the events dispatched during the `HttpKernel::handle` cycle. Typically a listener is a PHP class with a method that's executed, but it can be anything. For more information on creating and attaching event listeners, see *The EventDispatcher Component*.

The name of each of the "kernel" events is defined as a constant on the *KernelEvents*[44] class. Additionally, each event listener is passed a single argument, which is some sub-class of *KernelEvent*[45]. This object contains information about the current state of the system and each event has their own event object:

Name	KernelEvents Constant	Argument passed to the listener
kernel.request	KernelEvents::REQUEST	*GetResponseEvent*[46]
kernel.controller	KernelEvents::CONTROLLER	*FilterControllerEvent*[47]
kernel.view	KernelEvents::VIEW	*GetResponseForControllerResultEvent*[48]
kernel.response	KernelEvents::RESPONSE	*FilterResponseEvent*[49]
kernel.finish_request	KernelEvents::FINISH_REQUEST	*FinishRequestEvent*[50]

40. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
41. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Exception/FlattenException.html
42. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.html
43. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/ExceptionListener.html
44. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelEvents.html
45. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
46. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html
47. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html
48. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
49. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html
50. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/FinishRequestEvent.html

Name	KernelEvents Constant	Argument passed to the listener
kernel.terminate	KernelEvents::TERMINATE	*PostResponseEvent*[51]
kernel.exception	KernelEvents::EXCEPTION	*GetResponseForExceptionEvent*[52]

A full Working Example

When using the HttpKernel component, you're free to attach any listeners to the core events and use any controller resolver that implements the *ControllerResolverInterface*[53]. However, the HttpKernel component comes with some built-in listeners and a built-in ControllerResolver that can be used to create a working example:

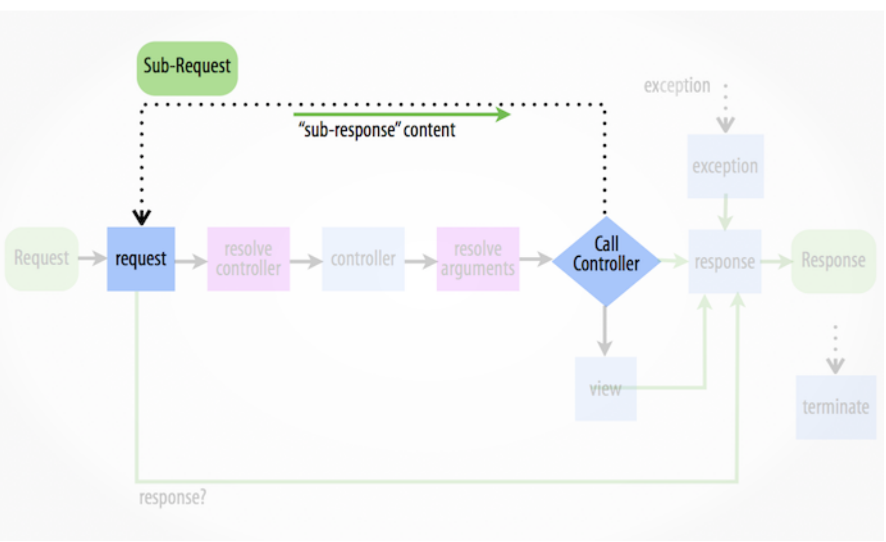
Listing 70-5

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RequestStack;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;
use Symfony\Component\HttpKernel\EventListener\RouterListener;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;

$routes = new RouteCollection();
$routes->add('hello', new Route('/hello/{name}', array(
        '_controller' => function (Request $request) {
            return new Response(
                sprintf("Hello %s", $request->get('name'))
            );
        }
    )
));

$request = Request::createFromGlobals();

$matcher = new UrlMatcher($routes, new RequestContext());

$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new RouterListener($matcher, new RequestStack()));

$resolver = new ControllerResolver();
$kernel = new HttpKernel($dispatcher, $resolver);

$response = $kernel->handle($request);
$response->send();

$kernel->terminate($request, $response);
```

51. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/PostResponseEvent.html
52. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
53. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html

Sub Requests

In addition to the "main" request that's sent into `HttpKernel::handle`, you can also send so-called "sub request". A sub request looks and acts like any other request, but typically serves to render just one small portion of a page instead of a full page. You'll most commonly make sub-requests from your controller (or perhaps from inside a template, that's being rendered by your controller).



To execute a sub request, use `HttpKernel::handle`, but change the second argument as follows:

Listing 70-6

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernelInterface;

// ...

// create some other request manually as needed
$request = new Request();
// for example, possibly set its _controller manually
$request->attributes->set('_controller', '...');

$response = $kernel->handle($request, HttpKernelInterface::SUB_REQUEST);
// do something with this response
```

This creates another full request-response cycle where this new `Request` is transformed into a `Response`. The only difference internally is that some listeners (e.g. security) may only act upon the master request. Each listener is passed some sub-class of *KernelEvent*[54], whose *isMasterRequest()*[55] can be used to check if the current request is a "master" or "sub" request.

For example, a listener that only needs to act on the master request may look like this:

Listing 70-7

```
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    if (!$event->isMasterRequest()) {
        return;
```

54. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
55. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/KernelEvent.html#method_isMasterRequest

```
8       }
9
10      // ...
11  }
```

Chapter 71
The Intl Component

A PHP replacement layer for the C *intl extension*[1] that also provides access to the localization data of the *ICU library*[2].

The replacement layer is limited to the locale "en". If you want to use other locales, you should *install the intl extension*[3] instead.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/intl` on *Packagist*[4]);
- Using the official Git repository (*https://github.com/symfony/intl*).

If you install the component via Composer, the following classes and functions of the intl extension will be automatically provided if the intl extension is not loaded:

- *`Collator`*[5]
- *`IntlDateFormatter`*[6]
- *`Locale`*[7]
- *`NumberFormatter`*[8]
- *`intl_error_name`*[9]

1. `http://www.php.net/manual/en/book.intl.php`
2. `http://site.icu-project.org/`
3. `http://www.php.net/manual/en/intl.setup.php`
4. `https://packagist.org/packages/symfony/intl`
5. `http://php.net/manual/en/class.collator.php`
6. `http://php.net/manual/en/class.intldateformatter.php`
7. `http://php.net/manual/en/class.locale.php`
8. `http://php.net/manual/en/class.numberformatter.php`

- *intl_is_failure*[10]
- *intl_get_error_code*[11]
- *intl_get_error_message*[12]

When the intl extension is not available, the following classes are used to replace the intl classes:

- *Collator*[13]
- *IntlDateFormatter*[14]
- *Locale*[15]
- *NumberFormatter*[16]
- *IntlGlobals*[17]

Composer automatically exposes these classes in the global namespace.

If you don't use Composer but the *Symfony ClassLoader component*, you need to expose them manually by adding the following lines to your autoload code:

Listing 71-1

```
if (!function_exists('intl_is_failure')) {
    require '/path/to/Icu/Resources/stubs/functions.php';

    $loader->registerPrefixFallback('/path/to/Icu/Resources/stubs');
}
```

Writing and Reading Resource Bundles

The *ResourceBundle*[18] class is not currently supported by this component. Instead, it includes a set of readers and writers for reading and writing arrays (or array-like objects) from/to resource bundle files. The following classes are supported:

- TextBundleWriter
- PhpBundleWriter
- BinaryBundleReader
- PhpBundleReader
- BufferedBundleReader
- StructuredBundleReader

Continue reading if you are interested in how to use these classes. Otherwise skip this section and jump to Accessing ICU Data.

TextBundleWriter

The *TextBundleWriter*[19] writes an array or an array-like object to a plain-text resource bundle. The resulting .txt file can be converted to a binary .res file with the *BundleCompiler*[20] class:

9. http://php.net/manual/en/function.intl-error-name.php
10. http://php.net/manual/en/function.intl-is-failure.php
11. http://php.net/manual/en/function.intl-get-error-code.php
12. http://php.net/manual/en/function.intl-get-error-message.php
13. http://api.symfony.com/3.0/Symfony/Component/Intl/Collator/Collator.html
14. http://api.symfony.com/3.0/Symfony/Component/Intl/DateFormatter/IntlDateFormatter.html
15. http://api.symfony.com/3.0/Symfony/Component/Intl/Locale/Locale.html
16. http://api.symfony.com/3.0/Symfony/Component/Intl/NumberFormatter/NumberFormatter.html
17. http://api.symfony.com/3.0/Symfony/Component/Intl/Globals/IntlGlobals.html
18. http://php.net/manual/en/class.resourcebundle.php
19. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Writer/TextBundleWriter.html
20. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html

Listing 71-2

```
use Symfony\Component\Intl\ResourceBundle\Writer\TextBundleWriter;
use Symfony\Component\Intl\ResourceBundle\Compiler\BundleCompiler;

$writer = new TextBundleWriter();
$writer->write('/path/to/bundle', 'en', array(
    'Data' => array(
        'entry1',
        'entry2',
        // ...
    ),
));

$compiler = new BundleCompiler();
$compiler->compile('/path/to/bundle', '/path/to/binary/bundle');
```

The command "genrb" must be available for the *BundleCompiler*[21] to work. If the command is located in a non-standard location, you can pass its path to the *BundleCompiler*[22] constructor.

PhpBundleWriter

The *PhpBundleWriter*[23] writes an array or an array-like object to a .php resource bundle:

Listing 71-3

```
use Symfony\Component\Intl\ResourceBundle\Writer\PhpBundleWriter;

$writer = new PhpBundleWriter();
$writer->write('/path/to/bundle', 'en', array(
    'Data' => array(
        'entry1',
        'entry2',
        // ...
    ),
));
```

BinaryBundleReader

The *BinaryBundleReader*[24] reads binary resource bundle files and returns an array or an array-like object. This class currently only works with the *intl extension*[25] installed:

Listing 71-4

```
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;

$reader = new BinaryBundleReader();
$data = $reader->read('/path/to/bundle', 'en');

var_dump($data['Data']['entry1']);
```

21. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html
22. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html
23. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Writer/PhpBundleWriter.html
24. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/BinaryBundleReader.html
25. http://www.php.net/manual/en/book.intl.php

PhpBundleReader

The *PhpBundleReader*[26] reads resource bundles from .php files and returns an array or an array-like object:

Listing 71-5
```
use Symfony\Component\Intl\ResourceBundle\Reader\PhpBundleReader;

$reader = new PhpBundleReader();
$data = $reader->read('/path/to/bundle', 'en');

var_dump($data['Data']['entry1']);
```

BufferedBundleReader

The *BufferedBundleReader*[27] wraps another reader, but keeps the last N reads in a buffer, where N is a buffer size passed to the constructor:

Listing 71-6
```
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;
use Symfony\Component\Intl\ResourceBundle\Reader\BufferedBundleReader;

$reader = new BufferedBundleReader(new BinaryBundleReader(), 10);

// actually reads the file
$data = $reader->read('/path/to/bundle', 'en');

// returns data from the buffer
$data = $reader->read('/path/to/bundle', 'en');

// actually reads the file
$data = $reader->read('/path/to/bundle', 'fr');
```

StructuredBundleReader

The *StructuredBundleReader*[28] wraps another reader and offers a *readEntry()*[29] method for reading an entry of the resource bundle without having to worry whether array keys are set or not. If a path cannot be resolved, `null` is returned:

Listing 71-7
```
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;
use Symfony\Component\Intl\ResourceBundle\Reader\StructuredBundleReader;

$reader = new StructuredBundleReader(new BinaryBundleReader());

$data = $reader->read('/path/to/bundle', 'en');

// Produces an error if the key "Data" does not exist
var_dump($data['Data']['entry1']);

// Returns null if the key "Data" does not exist
var_dump($reader->readEntry('/path/to/bundle', 'en', array('Data', 'entry1')));
```

26. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/PhpBundleReader.html
27. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/BufferedBundleReader.html
28. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReader.html
29. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReaderInterface.html#method_readEntry

Additionally, the *readEntry()*[30] method resolves fallback locales. For example, the fallback locale of "en_GB" is "en". For single-valued entries (strings, numbers etc.), the entry will be read from the fallback locale if it cannot be found in the more specific locale. For multi-valued entries (arrays), the values of the more specific and the fallback locale will be merged. In order to suppress this behavior, the last parameter `$fallback` can be set to `false`:

Listing 71-8
```
var_dump($reader->readEntry(
    '/path/to/bundle',
    'en',
    array('Data', 'entry1'),
    false
));
```

Accessing ICU Data

The ICU data is located in several "resource bundles". You can access a PHP wrapper of these bundles through the static *Intl*[31] class. At the moment, the following data is supported:

- Language and Script Names
- Country Names
- Locales
- Currencies

Language and Script Names

The translations of language and script names can be found in the language bundle:

Listing 71-9
```
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$languages = Intl::getLanguageBundle()->getLanguageNames();
// => array('ab' => 'Abkhazian', ...)

$language = Intl::getLanguageBundle()->getLanguageName('de');
// => 'German'

$language = Intl::getLanguageBundle()->getLanguageName('de', 'AT');
// => 'Austrian German'

$scripts = Intl::getLanguageBundle()->getScriptNames();
// => array('Arab' => 'Arabic', ...)

$script = Intl::getLanguageBundle()->getScriptName('Hans');
// => 'Simplified'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 71-10
```
$languages = Intl::getLanguageBundle()->getLanguageNames('de');
// => array('ab' => 'Abchasisch', ...)
```

30. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReaderInterface.html#method_readEntry
31. http://api.symfony.com/3.0/Symfony/Component/Intl/Intl.html

Country Names

The translations of country names can be found in the region bundle:

Listing 71-11
```
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$countries = Intl::getRegionBundle()->getCountryNames();
// => array('AF' => 'Afghanistan', ...)

$country = Intl::getRegionBundle()->getCountryName('GB');
// => 'United Kingdom'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 71-12
```
$countries = Intl::getRegionBundle()->getCountryNames('de');
// => array('AF' => 'Afghanistan', ...)
```

Locales

The translations of locale names can be found in the locale bundle:

Listing 71-13
```
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$locales = Intl::getLocaleBundle()->getLocaleNames();
// => array('af' => 'Afrikaans', ...)

$locale = Intl::getLocaleBundle()->getLocaleName('zh_Hans_MO');
// => 'Chinese (Simplified, Macau SAR China)'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 71-14
```
$locales = Intl::getLocaleBundle()->getLocaleNames('de');
// => array('af' => 'Afrikaans', ...)
```

Currencies

The translations of currency names and other currency-related information can be found in the currency bundle:

Listing 71-15
```
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$currencies = Intl::getCurrencyBundle()->getCurrencyNames();
// => array('AFN' => 'Afghan Afghani', ...)

$currency = Intl::getCurrencyBundle()->getCurrencyName('INR');
// => 'Indian Rupee'

$symbol = Intl::getCurrencyBundle()->getCurrencySymbol('INR');
// => '?'
```

```
13
14 $fractionDigits = Intl::getCurrencyBundle()->getFractionDigits('INR');
15 // => 2
16
17 $roundingIncrement = Intl::getCurrencyBundle()->getRoundingIncrement('INR');
18 // => 0
```

All methods (except for *getFractionDigits()*[32] and *getRoundingIncrement()*[33]) accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 71-16
```
$currencies = Intl::getCurrencyBundle()->getCurrencyNames('de');
// => array('AFN' => 'Afghanische Afghani', ...)
```

That's all you need to know for now. Have fun coding!

32. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/CurrencyBundleInterface.html#method_getFractionDigits
33. http://api.symfony.com/3.0/Symfony/Component/Intl/ResourceBundle/CurrencyBundleInterface.html#method_getRoundingIncrement

Chapter 72

The OptionsResolver Component

The OptionsResolver component is *array_replace*[1] on steroids. It allows you to create an options system with required options, defaults, validation (type, value), normalization and more.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/options-resolver` on *Packagist*[2]);
- Use the official Git repository (*https://github.com/symfony/options-resolver*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Notes on Previous Versions

Usage

Imagine you have a `Mailer` class which has four options: `host`, `username`, `password` and `port`:

Listing 72-1

```
class Mailer
{
    protected $options;

    public function __construct(array $options = array())
    {
```

1. http://php.net/manual/en/function.array-replace.php
2. https://packagist.org/packages/symfony/options-resolver

```
7             $this->options = $options;
8         }
9     }
```

When accessing the `$options`, you need to add a lot of boilerplate code to check which options are set:

Listing 72-2

```
1   class Mailer
2   {
3       // ...
4       public function sendMail($from, $to)
5       {
6           $mail = ...;
7
8           $mail->setHost(isset($this->options['host'])
9               ? $this->options['host']
10              : 'smtp.example.org');
11
12          $mail->setUsername(isset($this->options['username'])
13              ? $this->options['username']
14              : 'user');
15
16          $mail->setPassword(isset($this->options['password'])
17              ? $this->options['password']
18              : 'pa$$word');
19
20          $mail->setPort(isset($this->options['port'])
21              ? $this->options['port']
22              : 25);
23
24          // ...
25      }
26  }
```

This boilerplate is hard to read and repetitive. Also, the default values of the options are buried in the business logic of your code. Use the *array_replace*[3] to fix that:

Listing 72-3

```
1   class Mailer
2   {
3       // ...
4
5       public function __construct(array $options = array())
6       {
7           $this->options = array_replace(array(
8               'host'     => 'smtp.example.org',
9               'username' => 'user',
10              'password' => 'pa$$word',
11              'port'     => 25,
12          ), $options);
13      }
14  }
```

Now all four options are guaranteed to be set. But what happens if the user of the `Mailer` class makes a mistake?

Listing 72-4

3. http://php.net/manual/en/function.array-replace.php

```
1 $mailer = new Mailer(array(
2     'usernme' => 'johndoe',  // usernAme misspelled
3 ));
```

No error will be shown. In the best case, the bug will appear during testing, but the developer will spend time looking for the problem. In the worst case, the bug might not appear until it's deployed to the live system.

Fortunately, the *OptionsResolver*[4] class helps you to fix this problem:

Listing 72-5
```
1  use Symfony\Component\OptionsResolver\OptionsResolver;
2
3  class Mailer
4  {
5      // ...
6
7      public function __construct(array $options = array())
8      {
9          $resolver = new OptionsResolver();
10         $resolver->setDefaults(array(
11             'host'     => 'smtp.example.org',
12             'username' => 'user',
13             'password' => 'pa$$word',
14             'port'     => 25,
15         ));
16
17         $this->options = $resolver->resolve($options);
18     }
19 }
```

Like before, all options will be guaranteed to be set. Additionally, an *UndefinedOptionsException*[5] is thrown if an unknown option is passed:

Listing 72-6
```
1 $mailer = new Mailer(array(
2     'usernme' => 'johndoe',
3 ));
4
5 // UndefinedOptionsException: The option "usernme" does not exist.
6 // Known options are: "host", "password", "port", "username"
```

The rest of your code can access the values of the options without boilerplate code:

Listing 72-7
```
1  // ...
2  class Mailer
3  {
4      // ...
5
6      public function sendMail($from, $to)
7      {
8          $mail = ...;
9          $mail->setHost($this->options['host']);
10         $mail->setUsername($this->options['username']);
11         $mail->setPassword($this->options['password']);
12         $mail->setPort($this->options['port']);
```

4. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html
5. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/Exception/UndefinedOptionsException.html

```
13          // ...
14      }
15  }
```

It's a good practice to split the option configuration into a separate method:

Listing 72-8

```
1   // ...
2   class Mailer
3   {
4       // ...
5
6       public function __construct(array $options = array())
7       {
8           $resolver = new OptionsResolver();
9           $this->configureOptions($resolver);
10
11          $this->options = $resolver->resolve($options);
12      }
13
14      public function configureOptions(OptionsResolver $resolver)
15      {
16          $resolver->setDefaults(array(
17              'host'       => 'smtp.example.org',
18              'username'   => 'user',
19              'password'   => 'pa$$word',
20              'port'       => 25,
21              'encryption' => null,
22          ));
23      }
24  }
```

First, your code becomes easier to read, especially if the constructor does more than processing options. Second, sub-classes may now override the `configureOptions()` method to adjust the configuration of the options:

Listing 72-9

```
1   // ...
2   class GoogleMailer extends Mailer
3   {
4       public function configureOptions(OptionsResolver $resolver)
5       {
6           parent::configureOptions($resolver);
7
8           $resolver->setDefaults(array(
9               'host' => 'smtp.google.com',
10              'encryption' => 'ssl',
11          ));
12      }
13  }
```

Required Options

If an option must be set by the caller, pass that option to *setRequired()*[6]. For example, to make the `host` option required, you can do:

Listing 72-10

6. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setRequired

```
1  // ...
2  class Mailer
3  {
4      // ...
5
6      public function configureOptions(OptionsResolver $resolver)
7      {
8          // ...
9          $resolver->setRequired('host');
10     }
11 }
```

If you omit a required option, a *MissingOptionsException*[7] will be thrown:

Listing 72-11
```
$mailer = new Mailer();

// MissingOptionsException: The required option "host" is missing.
```

The *setRequired()*[8] method accepts a single name or an array of option names if you have more than one required option:

Listing 72-12
```
1  // ...
2  class Mailer
3  {
4      // ...
5
6      public function configureOptions(OptionsResolver $resolver)
7      {
8          // ...
9          $resolver->setRequired(array('host', 'username', 'password'));
10     }
11 }
```

Use *isRequired()*[9] to find out if an option is required. You can use *getRequiredOptions()*[10] to retrieve the names of all required options:

Listing 72-13
```
1  // ...
2  class GoogleMailer extends Mailer
3  {
4      public function configureOptions(OptionsResolver $resolver)
5      {
6          parent::configureOptions($resolver);
7
8          if ($resolver->isRequired('host')) {
9              // ...
10         }
11
12         $requiredOptions = $resolver->getRequiredOptions();
13     }
14 }
```

7. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/Exception/MissingOptionsException.html
8. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setRequired
9. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_isRequired
10. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_getRequiredOptions

If you want to check whether a required option is still missing from the default options, you can use *isMissing()*[11]. The difference between this and *isRequired()*[12] is that this method will return false if a required option has already been set:

Listing 72-14
```
// ...
class Mailer
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
        // ...
        $resolver->setRequired('host');
    }
}

// ...
class GoogleMailer extends Mailer
{
    public function configureOptions(OptionsResolver $resolver)
    {
        parent::configureOptions($resolver);

        $resolver->isRequired('host');
        // => true

        $resolver->isMissing('host');
        // => true

        $resolver->setDefault('host', 'smtp.google.com');

        $resolver->isRequired('host');
        // => true

        $resolver->isMissing('host');
        // => false
    }
}
```

The method *getMissingOptions()*[13] lets you access the names of all missing options.

Type Validation

You can run additional checks on the options to make sure they were passed correctly. To validate the types of the options, call *setAllowedTypes()*[14]:

Listing 72-15
```
// ...
class Mailer
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
```

11. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_isMissing
12. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_isRequired
13. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_getMissingOptions
14. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setAllowedTypes

```
 8          // ...
 9          $resolver->setAllowedTypes('host', 'string');
10          $resolver->setAllowedTypes('port', array('null', 'int'));
11      }
12  }
```

For each option, you can define either just one type or an array of acceptable types. You can pass any type for which an `is_<type>()` function is defined in PHP. Additionally, you may pass fully qualified class or interface names.

If you pass an invalid option now, an *InvalidOptionsException*[15] is thrown:

Listing 72-16

```
1  $mailer = new Mailer(array(
2      'host' => 25,
3  ));
4
5  // InvalidOptionsException: The option "host" with value "25" is
6  // expected to be of type "string"
```

In sub-classes, you can use *addAllowedTypes()*[16] to add additional allowed types without erasing the ones already set.

Value Validation

Some options can only take one of a fixed list of predefined values. For example, suppose the `Mailer` class has a `transport` option which can be one of `sendmail`, `mail` and `smtp`. Use the method *setAllowedValues()*[17] to verify that the passed option contains one of these values:

Listing 72-17

```
 1  // ...
 2  class Mailer
 3  {
 4      // ...
 5
 6      public function configureOptions(OptionsResolver $resolver)
 7      {
 8          // ...
 9          $resolver->setDefault('transport', 'sendmail');
10          $resolver->setAllowedValues('transport', array('sendmail', 'mail', 'smtp'));
11      }
12  }
```

If you pass an invalid transport, an *InvalidOptionsException*[18] is thrown:

Listing 72-18

```
1  $mailer = new Mailer(array(
2      'transport' => 'send-mail',
3  ));
4
5  // InvalidOptionsException: The option "transport" has the value
6  // "send-mail", but is expected to be one of "sendmail", "mail", "smtp"
```

For options with more complicated validation schemes, pass a closure which returns `true` for acceptable values and `false` for invalid values:

15. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/Exception/InvalidOptionsException.html
16. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_addAllowedTypes
17. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setAllowedValues
18. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/Exception/InvalidOptionsException.html

Listing 72-19

```
$resolver->setAllowedValues(array(
    // ...
    $resolver->setAllowedValues('transport', function ($value) {
        // return true or false
    });
));
```

In sub-classes, you can use *addAllowedValues()*[19] to add additional allowed values without erasing the ones already set.

Option Normalization

Sometimes, option values need to be normalized before you can use them. For instance, assume that the `host` should always start with `http://`. To do that, you can write normalizers. Normalizers are executed after validating an option. You can configure a normalizer by calling *setNormalizer()*[20]:

Listing 72-20

```
use Symfony\Component\OptionsResolver\Options;

// ...
class Mailer
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
        // ...

        $resolver->setNormalizer('host', function (Options $options, $value) {
            if ('http://' !== substr($value, 0, 7)) {
                $value = 'http://'.$value;
            }

            return $value;
        });
    }
}
```

The normalizer receives the actual `$value` and returns the normalized form. You see that the closure also takes an `$options` parameter. This is useful if you need to use other options during normalization:

Listing 72-21

```
// ...
class Mailer
{
    // ...
    public function configureOptions(OptionsResolver $resolver)
    {
        // ...
        $resolver->setNormalizer('host', function (Options $options, $value) {
            if (!in_array(substr($value, 0, 7), array('http://', 'https://'))) {
                if ('ssl' === $options['encryption']) {
                    $value = 'https://'.$value;
                } else {
                    $value = 'http://'.$value;
```

19. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_addAllowedValues
20. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setNormalizer

```
14                 }
15             }
16 
17             return $value;
18         });
19     }
20 }
```

Default Values that Depend on another Option

Suppose you want to set the default value of the `port` option based on the encryption chosen by the user of the `Mailer` class. More precisely, you want to set the port to `465` if SSL is used and to `25` otherwise.

You can implement this feature by passing a closure as the default value of the `port` option. The closure receives the options as argument. Based on these options, you can return the desired default value:

Listing 72-22

```
1  use Symfony\Component\OptionsResolver\Options;
2 
3  // ...
4  class Mailer
5  {
6      // ...
7      public function configureOptions(OptionsResolver $resolver)
8      {
9          // ...
10         $resolver->setDefault('encryption', null);
11 
12         $resolver->setDefault('port', function (Options $options) {
13             if ('ssl' === $options['encryption']) {
14                 return 465;
15             }
16 
17             return 25;
18         });
19     }
20 }
```

The argument of the callable must be type hinted as `Options`. Otherwise, the callable itself is considered as the default value of the option.

The closure is only executed if the `port` option isn't set by the user or overwritten in a sub-class.

A previously set default value can be accessed by adding a second argument to the closure:

Listing 72-23

```
1  // ...
2  class Mailer
3  {
4      // ...
5      public function configureOptions(OptionsResolver $resolver)
6      {
```

```
 7          // ...
 8          $resolver->setDefaults(array(
 9              'encryption' => null,
10              'host' => 'example.org',
11          ));
12      }
13  }
14
15  class GoogleMailer extends Mailer
16  {
17      public function configureOptions(OptionsResolver $resolver)
18      {
19          parent::configureOptions($resolver);
20
21          $options->setDefault('host', function (Options $options, $previousValue) {
22              if ('ssl' === $options['encryption']) {
23                  return 'secure.example.org'
24              }
25
26              // Take default value configured in the base class
27              return $previousValue;
28          });
29      }
30  }
```

As seen in the example, this feature is mostly useful if you want to reuse the default values set in parent classes in sub-classes.

Options without Default Values

In some cases, it is useful to define an option without setting a default value. This is useful if you need to know whether or not the user *actually* set an option or not. For example, if you set the default value for an option, it's not possible to know whether the user passed this value or if it simply comes from the default:

Listing 72-24

```
 1  // ...
 2  class Mailer
 3  {
 4      // ...
 5      public function configureOptions(OptionsResolver $resolver)
 6      {
 7          // ...
 8          $resolver->setDefault('port', 25);
 9      }
10
11      // ...
12      public function sendMail($from, $to)
13      {
14          // Is this the default value or did the caller of the class really
15          // set the port to 25?
16          if (25 === $this->options['port']) {
17              // ...
18          }
19      }
20  }
```

You can use `setDefined()`[21] to define an option without setting a default value. Then the option will only be included in the resolved options if it was actually passed to `resolve()`[22]:

Listing 72-25

```php
// ...
class Mailer
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
        // ...
        $resolver->setDefined('port');
    }

    // ...
    public function sendMail($from, $to)
    {
        if (array_key_exists('port', $this->options)) {
            echo 'Set!';
        } else {
            echo 'Not Set!';
        }
    }
}

$mailer = new Mailer();
$mailer->sendMail($from, $to);
// => Not Set!

$mailer = new Mailer(array(
    'port' => 25,
));
$mailer->sendMail($from, $to);
// => Set!
```

You can also pass an array of option names if you want to define multiple options in one go:

Listing 72-26

```php
// ...
class Mailer
{
    // ...
    public function configureOptions(OptionsResolver $resolver)
    {
        // ...
        $resolver->setDefined(array('port', 'encryption'));
    }
}
```

The methods *isDefined()*[23] and *getDefinedOptions()*[24] let you find out which options are defined:

Listing 72-27

```php
// ...
class GoogleMailer extends Mailer
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
```

21. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_setDefined
22. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_resolve
23. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_isDefined
24. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html#method_getDefinedOptions

```
7      {
8          parent::configureOptions($resolver);
9
10         if ($resolver->isDefined('host')) {
11             // One of the following was called:
12
13             // $resolver->setDefault('host', ...);
14             // $resolver->setRequired('host');
15             // $resolver->setDefined('host');
16         }
17
18         $definedOptions = $resolver->getDefinedOptions();
19     }
20 }
```

Performance Tweaks

With the current implementation, the `configureOptions()` method will be called for every single instance of the `Mailer` class. Depending on the amount of option configuration and the number of created instances, this may add noticeable overhead to your application. If that overhead becomes a problem, you can change your code to do the configuration only once per class:

Listing 72-28

```
1  // ...
2  class Mailer
3  {
4      private static $resolversByClass = array();
5
6      protected $options;
7
8      public function __construct(array $options = array())
9      {
10         // What type of Mailer is this, a Mailer, a GoogleMailer, ... ?
11         $class = get_class($this);
12
13         // Was configureOptions() executed before for this class?
14         if (!isset(self::$resolversByClass[$class])) {
15             self::$resolversByClass[$class] = new OptionsResolver();
16             $this->configureOptions(self::$resolversByClass[$class]);
17         }
18
19         $this->options = self::$resolversByClass[$class]->resolve($options);
20     }
21
22     public function configureOptions(OptionsResolver $resolver)
23     {
24         // ...
25     }
26 }
```

Now the *`OptionsResolver`*[25] instance will be created once per class and reused from that on. Be aware that this may lead to memory leaks in long-running applications, if the default options contain references to objects or object graphs. If that's the case for you, implement a method `clearOptionsConfig()` and call it periodically:

Listing 72-29

25. http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/OptionsResolver.html

```php
// ...
class Mailer
{
    private static $resolversByClass = array();

    public static function clearOptionsConfig()
    {
        self::$resolversByClass = array();
    }

    // ...
}
```

That's it! You now have all the tools and knowledge needed to easily process options in your code.

Chapter 73
The PHPUnit Bridge

> The PHPUnit Bridge provides utilities to report legacy tests and usage of deprecated code and a helper for time-sensitive tests.

It comes with the following features:

- Forces the tests to use a consistent locale (`C`);
- Auto-register `class_exists` to load Doctrine annotations (when used);
- It displays the whole list of deprecated features used in the application;
- Displays the stack trace of a deprecation on-demand;
- Provides a `ClockMock` helper class for time-sensitive tests.

New in version 2.7: The PHPUnit Bridge was introduced in Symfony 2.7. It is however possible to install the bridge in any Symfony application (even 2.3).

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/phpunit-bridge` on *Packagist*[1]); as a dev dependency;
- Use the official Git repository (*https://github.com/symfony/phpunit-bridge*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

1. https://packagist.org/packages/symfony/phpunit-bridge

Usage

Once the component installed, it automatically registers a *PHPUnit event listener*[2] which in turn registers a *PHP error handler*[3] called *DeprecationErrorHandler*[4]. After running your PHPUnit tests, you will get a report similar to this one:

```
$ phpunit -c app
PHPUnit 4.3.5 by Sebastian Bergmann.

Configuration read from /Users/javier/Desktop/symfony_demo_2/app/phpunit.xml.dist

.................

Time: 3.44 seconds, Memory: 49.75Mb

OK (17 tests, 21 assertions)

Remaining deprecation notices (2)

getEntityManager is deprecated since Symfony 2.1. Use getManager instead: 2x
    1x in DefaultControllerTest::testPublicUrls from AppBundle\Tests\Controller
    1x in BlogControllerTest::testIndex from AppBundle\Tests\Controller
```

The summary includes:

Unsilenced
: Reports deprecation notices that were triggered without the recommended *@-silencing operator*[5].

Legacy
: Deprecation notices denote tests that explicitly test some legacy features.

Remaining/Other
: Deprecation notices are all other (non-legacy) notices, grouped by message, test class and method.

Trigger Deprecation Notices

Deprecation notices can be triggered by using:

Listing 73-1
```
@trigger_error('Your deprecation message', E_USER_DEPRECATED);
```

Without the *@-silencing operator*[6], users would need to opt-out from deprecation notices. Silencing by default swaps this behavior and allows users to opt-in when they are ready to cope with them (by adding a custom error handler like the one provided by this bridge). When not silenced, deprecation notices will appear in the **Unsilenced** section of the deprecation report.

Mark Tests as Legacy

There are four ways to mark a test as legacy:

2. https://phpunit.de/manual/current/en/extending-phpunit.html#extending-phpunit.PHPUnit_Framework_TestListener
3. http://php.net/manual/en/book.errorfunc.php
4. http://api.symfony.com/3.0/Symfony/Bridge/PhpUnit/DeprecationErrorHandler.html
5. http://php.net/manual/en/language.operators.errorcontrol.php
6. http://php.net/manual/en/language.operators.errorcontrol.php

- (**Recommended**) Add the `@group legacy` annotation to its class or method;
- Make its class name start with the `Legacy` prefix;
- Make its method name start with `testLegacy` instead of `test`;
- Make its data provider start with `provideLegacy` or `getLegacy`.

Configuration

In case you need to inspect the stack trace of a particular deprecation triggered by your unit tests, you can set the `SYMFONY_DEPRECATIONS_HELPER` *environment variable*[7] to a regular expression that matches this deprecation's message, enclosed with `/`. For example, with:

Listing 73-2

```
<!-- http://phpunit.de/manual/4.1/en/appendixes.configuration.html -->
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.1/phpunit.xsd"
>

    <!-- ... -->

    <php>
        <server name="KERNEL_DIR" value="app/" />
        <env name="SYMFONY_DEPRECATIONS_HELPER" value="/foobar/" />
    </php>
</phpunit>
```

PHPUnit[8] will stop your test suite once a deprecation notice is triggered whose message contains the `"foobar"` string.

Making Tests Fail

By default, any non-legacy-tagged or any non-*@-silenced*[9] deprecation notices will make tests fail. Alternatively, setting `SYMFONY_DEPRECATIONS_HELPER` to an arbitrary value (ex: `320`) will make the tests fails only if a higher number of deprecation notices is reached (`0` is the default value). You can also set the value `"weak"` which will make the bridge ignore any deprecation notices. This is useful to projects that must use deprecated interfaces for backward compatibility reasons.

Time-sensitive Tests

Use Case

If you have this kind of time-related tests:

Listing 73-3

```
use Symfony\Component\Stopwatch\Stopwatch;

class MyTest extends \PHPUnit_Framework_TestCase
{
    public function testSomething()
    {
```

7. https://phpunit.de/manual/current/en/appendixes.configuration.html#appendixes.configuration.php-ini-constants-variables
8. https://phpunit.de
9. http://php.net/manual/en/language.operators.errorcontrol.php

```
7         $stopwatch = new Stopwatch();
8
9         $stopwatch->start();
10        sleep(10);
11        $duration = $stopwatch->stop();
12
13        $this->assertEquals(10, $duration);
14    }
15 }
```

You used the *Symfony Stopwatch Component* to calculate the duration time of your process, here 10 seconds. However, depending on the load of the server your the processes running on your local machine, the `$duration` could for example be *10.000023s* instead of *10s*.

This kind of tests are called transient tests: they are failing randomly depending on spurious and external circumstances. They are often cause trouble when using public continuous integration services like *Travis CI*[10].

Clock Mocking

The *ClockMock*[11] class provided by this bridge allows you to mock the PHP's built-in time functions `time()`, `microtime()`, `sleep()` and `usleep()`.

To use the `ClockMock` class in your test, you can:

- (**Recommended**) Add the `@group time-sensitive` annotation to its class or method;
- Register it manually by calling `ClockMock::register(__CLASS__)` and `ClockMock::withClockMock(true)` before the test and `ClockMock::withClockMock(false)` after the test.

As a result, the following is guarenteed to work and is no longer a transient test:

Listing 73-4

```
1  use Symfony\Component\Stopwatch\Stopwatch;
2
3  /**
4   * @group time-sensitive
5   */
6  class MyTest extends \PHPUnit_Framework_TestCase
7  {
8      public function testSomething()
9      {
10         $stopwatch = new Stopwatch();
11
12         $stopwatch->start();
13         sleep(10);
14         $duration = $stopwatch->stop();
15
16         $this->assertEquals(10, $duration);
17     }
18 }
```

And that's all!

10. https://travis-ci.com/
11. http://api.symfony.com/3.0/Symfony/Bridge/PhpUnit/ClockMock.html

An added bonus of using the `ClockMock` class is that time passes instantly. Using PHP's `sleep(10)` will make your test wait for 10 actual seconds (more or less). In contrast, the `ClockMock` class advances the internal clock the given number of seconds without actually waiting that time, so your test will execute 10 seconds faster.

Troubleshooting

The `@group time-sensitive` works "by convention" and assumes that the namespace of the tested class can be obtained just by removing the `\Tests\` part from the test namespace. I.e. that if the your test case fully-qualified class name (FQCN) is `App\Tests\Watch\DummyWatchTest`, it assumes the tested class FQCN is `App\Watch\DummyWatch`.

If this convention doesn't work for your application, you can also configure the mocked namespaces in the `phpunit.xml` file, as done for example in the *HttpKernel Component*:

Listing 73-5

```
<!-- http://phpunit.de/manual/4.1/en/appendixes.configuration.html -->
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.1/phpunit.xsd"
>

    <!-- ... -->

    <listeners>
        <listener class="Symfony\Bridge\PhpUnit\SymfonyTestsListener">
            <arguments>
                <array>
                    <element><string>Symfony\Component\HttpFoundation</string></element>
                </array>
            </arguments>
        </listener>
    </listeners>
</phpunit>
```

Chapter 74

The Process Component

> The Process component executes commands in sub-processes.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/process` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/process*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The *`Process`*[2] class allows you to execute a command in a sub-process:

Listing 74-1

```
use Symfony\Component\Process\Process;
use Symfony\Component\Process\Exception\ProcessFailedException;

$process = new Process('ls -lsa');
$process->run();

// executes after the command finishes
if (!$process->isSuccessful()) {
    throw new ProcessFailedException($process);
}
```

1. https://packagist.org/packages/symfony/process
2. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html

```
11
12 echo $process->getOutput();
```

The component takes care of the subtle differences between the different platforms when executing the command.

The `getOutput()` method always returns the whole content of the standard output of the command and `getErrorOutput()` the content of the error output. Alternatively, the *getIncrementalOutput()*[3] and *getIncrementalErrorOutput()*[4] methods returns the new outputs since the last call.

The *clearOutput()*[5] method clears the contents of the output and *clearErrorOutput()*[6] clears the contents of the error output.

The `mustRun()` method is identical to `run()`, except that it will throw a *ProcessFailedException*[7] if the process couldn't be executed successfully (i.e. the process exited with a non-zero code):

Listing 74-2

```
1  use Symfony\Component\Process\Exception\ProcessFailedException;
2  use Symfony\Component\Process\Process;
3
4  $process = new Process('ls -lsa');
5
6  try {
7      $process->mustRun();
8
9      echo $process->getOutput();
10 } catch (ProcessFailedException $e) {
11     echo $e->getMessage();
12 }
```

Getting real-time Process Output

When executing a long running command (like rsync-ing files to a remote server), you can give feedback to the end user in real-time by passing an anonymous function to the *run()*[8] method:

Listing 74-3

```
1  use Symfony\Component\Process\Process;
2
3  $process = new Process('ls -lsa');
4  $process->run(function ($type, $buffer) {
5      if (Process::ERR === $type) {
6          echo 'ERR > '.$buffer;
7      } else {
8          echo 'OUT > '.$buffer;
9      }
10 });
```

3. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_getIncrementalOutput
4. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_getIncrementalErrorOutput
5. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_clearOutput
6. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_clearErrorOutput
7. http://api.symfony.com/3.0/Symfony/Component/Process/Exception/ProcessFailedException.html
8. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_run

Running Processes Asynchronously

You can also start the subprocess and then let it run asynchronously, retrieving output and the status in your main process whenever you need it. Use the *start()*[9] method to start an asynchronous process, the *isRunning()*[10] method to check if the process is done and the *getOutput()*[11] method to get the output:

Listing 74-4
```
1 $process = new Process('ls -lsa');
2 $process->start();
3
4 while ($process->isRunning()) {
5     // waiting for process to finish
6 }
7
8 echo $process->getOutput();
```

You can also wait for a process to end if you started it asynchronously and are done doing other stuff:

Listing 74-5
```
1  $process = new Process('ls -lsa');
2  $process->start();
3
4  // ... do other things
5
6  $process->wait(function ($type, $buffer) {
7      if (Process::ERR === $type) {
8          echo 'ERR > '.$buffer;
9      } else {
10         echo 'OUT > '.$buffer;
11     }
12 });
```

The *wait()*[12] method is blocking, which means that your code will halt at this line until the external process is completed.

Stopping a Process

Any asynchronous process can be stopped at any time with the *stop()*[13] method. This method takes two arguments: a timeout and a signal. Once the timeout is reached, the signal is sent to the running process. The default signal sent to a process is `SIGKILL`. Please read the signal documentation below to find out more about signal handling in the Process component:

Listing 74-6
```
1 $process = new Process('ls -lsa');
2 $process->start();
3
4 // ... do other things
5
6 $process->stop(3, SIGINT);
```

9. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_start
10. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_isRunning
11. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_getOutput
12. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_wait
13. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_stop

Executing PHP Code in Isolation

If you want to execute some PHP code in isolation, use the `PhpProcess` instead:

Listing 74-7
```
1 use Symfony\Component\Process\PhpProcess;
2
3 $process = new PhpProcess(<<<EOF
4     <?php echo 'Hello World'; ?>
5 EOF
6 );
7 $process->run();
```

To make your code work better on all platforms, you might want to use the *`ProcessBuilder`*[14] class instead:

Listing 74-8
```
use Symfony\Component\Process\ProcessBuilder;

$builder = new ProcessBuilder(array('ls', '-lsa'));
$builder->getProcess()->run();
```

In case you are building a binary driver, you can use the *`setPrefix()`*[15] method to prefix all the generated process commands.

The following example will generate two process commands for a tar binary adapter:

Listing 74-9
```
 1 use Symfony\Component\Process\ProcessBuilder;
 2
 3 $builder = new ProcessBuilder();
 4 $builder->setPrefix('/usr/bin/tar');
 5
 6 // '/usr/bin/tar' '--list' '--file=archive.tar.gz'
 7 echo $builder
 8     ->setArguments(array('--list', '--file=archive.tar.gz'))
 9     ->getProcess()
10     ->getCommandLine();
11
12 // '/usr/bin/tar' '-xzf' 'archive.tar.gz'
13 echo $builder
14     ->setArguments(array('-xzf', 'archive.tar.gz'))
15     ->getProcess()
16     ->getCommandLine();
```

Process Timeout

You can limit the amount of time a process takes to complete by setting a timeout (in seconds):

Listing 74-10
```
1 use Symfony\Component\Process\Process;
2
3 $process = new Process('ls -lsa');
4 $process->setTimeout(3600);
5 $process->run();
```

If the timeout is reached, a *`RuntimeException`*[16] is thrown.

14. http://api.symfony.com/3.0/Symfony/Component/Process/ProcessBuilder.html
15. http://api.symfony.com/3.0/Symfony/Component/Process/ProcessBuilder.html#method_setPrefix

For long running commands, it is your responsibility to perform the timeout check regularly:

Listing 74-11
```
$process->setTimeout(3600);
$process->start();

while ($condition) {
    // ...

    // check if the timeout is reached
    $process->checkTimeout();

    usleep(200000);
}
```

Process Idle Timeout

In contrast to the timeout of the previous paragraph, the idle timeout only considers the time since the last output was produced by the process:

Listing 74-12
```
use Symfony\Component\Process\Process;

$process = new Process('something-with-variable-runtime');
$process->setTimeout(3600);
$process->setIdleTimeout(60);
$process->run();
```

In the case above, a process is considered timed out, when either the total runtime exceeds 3600 seconds, or the process does not produce any output for 60 seconds.

Process Signals

When running a program asynchronously, you can send it POSIX signals with the *`signal()`*[17] method:

Listing 74-13
```
use Symfony\Component\Process\Process;

$process = new Process('find / -name "rabbit"');
$process->start();

// will send a SIGKILL to the process
$process->signal(SIGKILL);
```

Due to some limitations in PHP, if you're using signals with the Process component, you may have to prefix your commands with *exec*[18]. Please read *Symfony Issue#5759*[19] and *PHP Bug#39992*[20] to understand why this is happening.

POSIX signals are not available on Windows platforms, please refer to the *PHP documentation*[21] for available signals.

16. http://api.symfony.com/3.0/Symfony/Component/Process/Exception/RuntimeException.html
17. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_signal
18. https://en.wikipedia.org/wiki/Exec_(operating_system)
19. https://github.com/symfony/symfony/issues/5759
20. https://bugs.php.net/bug.php?id=39992

Process Pid

You can access the *pid*[22] of a running process with the *getPid()*[23] method.

Listing 74-14
```
1  use Symfony\Component\Process\Process;
2
3  $process = new Process('/usr/bin/php worker.php');
4  $process->start();
5
6  $pid = $process->getPid();
```

Due to some limitations in PHP, if you want to get the pid of a symfony Process, you may have to prefix your commands with *exec*[24]. Please read *Symfony Issue#5759*[25] to understand why this is happening.

Disabling Output

As standard output and error output are always fetched from the underlying process, it might be convenient to disable output in some cases to save memory. Use *disableOutput()*[26] and *enableOutput()*[27] to toggle this feature:

Listing 74-15
```
1  use Symfony\Component\Process\Process;
2
3  $process = new Process('/usr/bin/php worker.php');
4  $process->disableOutput();
5  $process->run();
```

You can not enable or disable the output while the process is running.

If you disable the output, you cannot access `getOutput`, `getIncrementalOutput`, `getErrorOutput` or `getIncrementalErrorOutput`. Moreover, you could not pass a callback to the `start`, `run` or `mustRun` methods or use `setIdleTimeout`.

21. http://php.net/manual/en/pcntl.constants.php
22. https://en.wikipedia.org/wiki/Process_identifier
23. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_getPid
24. https://en.wikipedia.org/wiki/Exec_(operating_system)
25. https://github.com/symfony/symfony/issues/5759
26. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_disableOutput
27. http://api.symfony.com/3.0/Symfony/Component/Process/Process.html#method_enableOutput

Chapter 75

The PropertyAccess Component

> The PropertyAccess component provides function to read and write from/to an object or array using a simple string notation.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/property-access` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/property-access*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The entry point of this component is the *`PropertyAccess::createPropertyAccessor`*[2] factory. This factory will create a new instance of the *`PropertyAccessor`*[3] class with the default configuration:

Listing 75-1

```
use Symfony\Component\PropertyAccess\PropertyAccess;

$accessor = PropertyAccess::createPropertyAccessor();
```

1. https://packagist.org/packages/symfony/property-access
2. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccess.html#method_createPropertyAccessor
3. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html

Reading from Arrays

You can read an array with the *PropertyAccessor::getValue*[4] method. This is done using the index notation that is used in PHP:

Listing 75-2
```
// ...
$person = array(
    'first_name' => 'Wouter',
);

var_dump($accessor->getValue($person, '[first_name]')); // 'Wouter'
var_dump($accessor->getValue($person, '[age]')); // null
```

As you can see, the method will return `null` if the index does not exists.

You can also use multi dimensional arrays:

Listing 75-3
```
// ...
$persons = array(
    array(
        'first_name' => 'Wouter',
    ),
    array(
        'first_name' => 'Ryan',
    )
);

var_dump($accessor->getValue($persons, '[0][first_name]')); // 'Wouter'
var_dump($accessor->getValue($persons, '[1][first_name]')); // 'Ryan'
```

Reading from Objects

The `getValue` method is a very robust method, and you can see all of its features when working with objects.

Accessing public Properties

To read from properties, use the "dot" notation:

Listing 75-4
```
// ...
$person = new Person();
$person->firstName = 'Wouter';

var_dump($accessor->getValue($person, 'firstName')); // 'Wouter'

$child = new Person();
$child->firstName = 'Bar';
$person->children = array($child);

var_dump($accessor->getValue($person, 'children[0].firstName')); // 'Bar'
```

4. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_getValue

Accessing public properties is the last option used by `PropertyAccessor`. It tries to access the value using the below methods first before using the property directly. For example, if you have a public property that has a getter method, it will use the getter.

Using Getters

The `getValue` method also supports reading using getters. The method will be created using common naming conventions for getters. It camelizes the property name (`first_name` becomes `FirstName`) and prefixes it with `get`. So the actual method becomes `getFirstName`:

Listing 75-5

```
// ...
class Person
{
    private $firstName = 'Wouter';

    public function getFirstName()
    {
        return $this->firstName;
    }
}

$person = new Person();

var_dump($accessor->getValue($person, 'first_name')); // 'Wouter'
```

Using Hassers/Issers

And it doesn't even stop there. If there is no getter found, the accessor will look for an isser or hasser. This method is created using the same way as getters, this means that you can do something like this:

Listing 75-6

```
// ...
class Person
{
    private $author = true;
    private $children = array();

    public function isAuthor()
    {
        return $this->author;
    }

    public function hasChildren()
    {
        return 0 !== count($this->children);
    }
}

$person = new Person();

if ($accessor->getValue($person, 'author')) {
    var_dump('He is an author');
}
if ($accessor->getValue($person, 'children')) {
    var_dump('He has children');
}
```

This will produce: `He is an author`

Magic `__get()` Method

The `getValue` method can also use the magic `__get` method:

Listing 75-7

```
// ...
class Person
{
    private $children = array(
        'Wouter' => array(...),
    );

    public function __get($id)
    {
        return $this->children[$id];
    }
}

$person = new Person();

var_dump($accessor->getValue($person, 'Wouter')); // array(...)
```

Magic `__call()` Method

At last, `getValue` can use the magic `__call` method, but you need to enable this feature by using *PropertyAccessorBuilder*[5]:

Listing 75-8

```
// ...
class Person
{
    private $children = array(
        'wouter' => array(...),
    );

    public function __call($name, $args)
    {
        $property = lcfirst(substr($name, 3));
        if ('get' === substr($name, 0, 3)) {
            return isset($this->children[$property])
                ? $this->children[$property]
                : null;
        } elseif ('set' === substr($name, 0, 3)) {
            $value = 1 == count($args) ? $args[0] : null;
            $this->children[$property] = $value;
        }
    }
}

$person = new Person();

// Enable magic __call
$accessor = PropertyAccess::createPropertyAccessorBuilder()
    ->enableMagicCall()
    ->getPropertyAccessor();
```

5. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html

```
28
29 var_dump($accessor->getValue($person, 'wouter')); // array(...)
```

The `__call` feature is disabled by default, you can enable it by calling *PropertyAccessorBuilder::enableMagicCall*[6] see Enable other Features.

Writing to Arrays

The `PropertyAccessor` class can do more than just read an array, it can also write to an array. This can be achieved using the *PropertyAccessor::setValue*[7] method:

Listing 75-9

```
1 // ...
2 $person = array();
3
4 $accessor->setValue($person, '[first_name]', 'Wouter');
5
6 var_dump($accessor->getValue($person, '[first_name]')); // 'Wouter'
7 // or
8 // var_dump($person['first_name']); // 'Wouter'
```

Writing to Objects

The `setValue` method has the same features as the `getValue` method. You can use setters, the magic `__set` method or properties to set values:

Listing 75-10

```
1 // ...
2 class Person
3 {
4     public $firstName;
5     private $lastName;
6     private $children = array();
7
8     public function setLastName($name)
9     {
10         $this->lastName = $name;
11     }
12
13     public function __set($property, $value)
14     {
15         $this->$property = $value;
16     }
17
18     // ...
19 }
20
21 $person = new Person();
22
```

6. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html#method_enableMagicCall
7. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_setValue

```
23 $accessor->setValue($person, 'firstName', 'Wouter');
24 $accessor->setValue($person, 'lastName', 'de Jong');
25 $accessor->setValue($person, 'children', array(new Person()));
26
27 var_dump($person->firstName); // 'Wouter'
28 var_dump($person->getLastName()); // 'de Jong'
29 var_dump($person->children); // array(Person());
```

You can also use `__call` to set values but you need to enable the feature, see Enable other Features.

Listing 75-11
```
1 // ...
2 class Person
3 {
4     private $children = array();
5
6     public function __call($name, $args)
7     {
8         $property = lcfirst(substr($name, 3));
9         if ('get' === substr($name, 0, 3)) {
10            return isset($this->children[$property])
11                ? $this->children[$property]
12                : null;
13        } elseif ('set' === substr($name, 0, 3)) {
14            $value = 1 == count($args) ? $args[0] : null;
15            $this->children[$property] = $value;
16        }
17    }
18
19 }
20
21 $person = new Person();
22
23 // Enable magic __call
24 $accessor = PropertyAccess::createPropertyAccessorBuilder()
25     ->enableMagicCall()
26     ->getPropertyAccessor();
27
28 $accessor->setValue($person, 'wouter', array(...));
29
30 var_dump($person->getWouter()); // array(...)
```

Checking Property Paths

When you want to check whether *`PropertyAccessor::getValue`*[8] can safely be called without actually calling that method, you can use *`PropertyAccessor::isReadable`*[9] instead:

Listing 75-12
```
1 $person = new Person();
2
3 if ($accessor->isReadable($person, 'firstName')) {
4     // ...
5 }
```

8. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_getValue
9. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_isReadable

The same is possible for *PropertyAccessor::setValue*[10]: Call the *PropertyAccessor::isWritable*[11] method to find out whether a property path can be updated:

Listing 75-13
```
$person = new Person();

if ($accessor->isWritable($person, 'firstName')) {
    // ...
}
```

Mixing Objects and Arrays

You can also mix objects and arrays:

Listing 75-14
```
// ...
class Person
{
    public $firstName;
    private $children = array();

    public function setChildren($children)
    {
        $this->children = $children;
    }

    public function getChildren()
    {
        return $this->children;
    }
}

$person = new Person();

$accessor->setValue($person, 'children[0]', new Person);
// equal to $person->getChildren()[0] = new Person()

$accessor->setValue($person, 'children[0].firstName', 'Wouter');
// equal to $person->getChildren()[0]->firstName = 'Wouter'

var_dump('Hello '.$accessor->getValue($person, 'children[0].firstName')); // 'Wouter'
// equal to $person->getChildren()[0]->firstName
```

Enable other Features

The *PropertyAccessor*[12] can be configured to enable extra features. To do that you could use the *PropertyAccessorBuilder*[13]:

Listing 75-15
```
// ...
$accessorBuilder = PropertyAccess::createPropertyAccessorBuilder();

// Enable magic __call
```

10. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_setValue
11. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html#method_isWritable
12. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessor.html
13. http://api.symfony.com/3.0/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html

```
5  $accessorBuilder->enableMagicCall();
6
7  // Disable magic __call
8  $accessorBuilder->disableMagicCall();
9
10 // Check if magic __call handling is enabled
11 $accessorBuilder->isMagicCallEnabled(); // true or false
12
13 // At the end get the configured property accessor
14 $accessor = $accessorBuilder->getPropertyAccessor();
15
16 // Or all in one
17 $accessor = PropertyAccess::createPropertyAccessorBuilder()
18     ->enableMagicCall()
19     ->getPropertyAccessor();
```

Or you can pass parameters directly to the constructor (not the recommended way):

Listing 75-16
```
// ...
$accessor = new PropertyAccessor(true); // this enables handling of magic __call
```

Chapter 76

The Routing Component

> The Routing component maps an HTTP request to a set of configuration variables.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/routing` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/routing*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

In order to set up a basic routing system you need three parts:

- A *RouteCollection*[2], which contains the route definitions (instances of the class *Route*[3])
- A *RequestContext*[4], which has information about the request
- A *UrlMatcher*[5], which performs the mapping of the request to a single route

Here is a quick example. Notice that this assumes that you've already configured your autoloader to load the Routing component:

Listing 76-1

1. https://packagist.org/packages/symfony/routing
2. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html
3. http://api.symfony.com/3.0/Symfony/Component/Routing/Route.html
4. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
5. http://api.symfony.com/3.0/Symfony/Component/Routing/Matcher/UrlMatcher.html

```php
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$route = new Route('/foo', array('controller' => 'MyController'));
$routes = new RouteCollection();
$routes->add('route_name', $route);

$context = new RequestContext('/');

$matcher = new UrlMatcher($routes, $context);

$parameters = $matcher->match('/foo');
// array('controller' => 'MyController', '_route' => 'route_name')
```

The *RequestContext*[6] parameters can be populated with the values stored in `$_SERVER`, but it's easier to use the HttpFoundation component as explained below.

You can add as many routes as you like to a *RouteCollection*[7].

The *RouteCollection::add()*[8] method takes two arguments. The first is the name of the route. The second is a *Route*[9] object, which expects a URL path and some array of custom variables in its constructor. This array of custom variables can be *anything* that's significant to your application, and is returned when that route is matched.

The *UrlMatcher::match()*[10] returns the variables you set on the route as well as the wildcard placeholders (see below). Your application can now use this information to continue processing the request. In addition to the configured variables, a `_route` key is added, which holds the name of the matched route.

If no matching route can be found, a *ResourceNotFoundException*[11] will be thrown.

Defining Routes

A full route definition can contain up to seven parts:

1. The URL path route. This is matched against the URL passed to the *RequestContext*, and can contain named wildcard placeholders (e.g. `{placeholders}`) to match dynamic parts in the URL.
2. An array of default values. This contains an array of arbitrary values that will be returned when the request matches the route.
3. An array of requirements. These define constraints for the values of the placeholders as regular expressions.
4. An array of options. These contain internal settings for the route and are the least commonly needed.
5. A host. This is matched against the host of the request. See *How to Match a Route Based on the Host* for more details.
6. An array of schemes. These enforce a certain HTTP scheme (`http`, `https`).
7. An array of methods. These enforce a certain HTTP request method (`HEAD`, `GET`, `POST`, ...).

6. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
7. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html
8. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html#method_add
9. http://api.symfony.com/3.0/Symfony/Component/Routing/Route.html
10. http://api.symfony.com/3.0/Symfony/Component/Routing/UrlMatcher.html#method_match
11. http://api.symfony.com/3.0/Symfony/Component/Routing/Exception/ResourceNotFoundException.html

Take the following route, which combines several of these ideas:

Listing 76-2

```
$route = new Route(
    '/archive/{month}', // path
    array('controller' => 'showArchive'), // default values
    array('month' => '[0-9]{4}-[0-9]{2}', 'subdomain' => 'www|m'), // requirements
    array(), // options
    '{subdomain}.example.com', // host
    array(), // schemes
    array() // methods
);

// ...

$parameters = $matcher->match('/archive/2012-01');
// array(
//     'controller' => 'showArchive',
//     'month' => '2012-01',
//     'subdomain' => 'www',
//     '_route' => ...
//  )

$parameters = $matcher->match('/archive/foo');
// throws ResourceNotFoundException
```

In this case, the route is matched by `/archive/2012-01`, because the `{month}` wildcard matches the regular expression wildcard given. However, `/archive/foo` does *not* match, because "foo" fails the month wildcard.

When using wildcards, these are returned in the array result when calling `match`. The part of the path that the wildcard matched (e.g. `2012-01`) is used as value.

If you want to match all URLs which start with a certain path and end in an arbitrary suffix you can use the following route definition:

Listing 76-3

```
$route = new Route(
    '/start/{suffix}',
    array('suffix' => ''),
    array('suffix' => '.*')
);
```

Using Prefixes

You can add routes or other instances of *`RouteCollection`*[12] to *another* collection. This way you can build a tree of routes. Additionally you can define a prefix and default values for the parameters, requirements, options, schemes and the host to all routes of a subtree using methods provided by the `RouteCollection` class:

Listing 76-4

```
$rootCollection = new RouteCollection();

$subCollection = new RouteCollection();
$subCollection->add(...);
$subCollection->add(...);
$subCollection->addPrefix('/prefix');
```

12. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html

```
7 $subCollection->addDefaults(array(...));
8 $subCollection->addRequirements(array(...));
9 $subCollection->addOptions(array(...));
10 $subCollection->setHost('admin.example.com');
11 $subCollection->setMethods(array('POST'));
12 $subCollection->setSchemes(array('https'));
13
14 $rootCollection->addCollection($subCollection);
```

Set the Request Parameters

The *RequestContext*[13] provides information about the current request. You can define all parameters of an HTTP request with this class via its constructor:

Listing 76-5

```
1 public function __construct(
2     $baseUrl = '',
3     $method = 'GET',
4     $host = 'localhost',
5     $scheme = 'http',
6     $httpPort = 80,
7     $httpsPort = 443,
8     $path = '/',
9     $queryString = ''
10 )
```

Normally you can pass the values from the `$_SERVER` variable to populate the *RequestContext*[14]. But If you use the *HttpFoundation* component, you can use its *Request*[15] class to feed the *RequestContext*[16] in a shortcut:

Listing 76-6

```
use Symfony\Component\HttpFoundation\Request;

$context = new RequestContext();
$context->fromRequest(Request::createFromGlobals());
```

Generate a URL

While the *UrlMatcher*[17] tries to find a route that fits the given request you can also build a URL from a certain route:

Listing 76-7

```
1 use Symfony\Component\Routing\Generator\UrlGenerator;
2
3 $routes = new RouteCollection();
4 $routes->add('show_post', new Route('/show/{slug}'));
5
6 $context = new RequestContext('/');
7
8 $generator = new UrlGenerator($routes, $context);
9
10 $url = $generator->generate('show_post', array(
11     'slug' => 'my-blog-post',
```

13. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
14. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
16. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
17. http://api.symfony.com/3.0/Symfony/Component/Routing/Matcher/UrlMatcher.html

```
12 ));
13 // /show/my-blog-post
```

If you have defined a scheme, an absolute URL is generated if the scheme of the current *RequestContext*[18] does not match the requirement.

Load Routes from a File

You've already seen how you can easily add routes to a collection right inside PHP. But you can also load routes from a number of different files.

The Routing component comes with a number of loader classes, each giving you the ability to load a collection of route definitions from an external file of some format. Each loader expects a *FileLocator*[19] instance as the constructor argument. You can use the *FileLocator*[20] to define an array of paths in which the loader will look for the requested files. If the file is found, the loader returns a *RouteCollection*[21].

If you're using the `YamlFileLoader`, then route definitions look like this:

Listing 76-8

```
1 # routes.yml
2 route1:
3     path:     /foo
4     defaults: { _controller: 'MyController::fooAction' }
5
6 route2:
7     path:     /foo/bar
8     defaults: { _controller: 'MyController::foobarAction' }
```

To load this file, you can use the following code. This assumes that your `routes.yml` file is in the same directory as the below code:

Listing 76-9

```
1 use Symfony\Component\Config\FileLocator;
2 use Symfony\Component\Routing\Loader\YamlFileLoader;
3
4 // look inside *this* directory
5 $locator = new FileLocator(array(__DIR__));
6 $loader = new YamlFileLoader($locator);
7 $collection = $loader->load('routes.yml');
```

Besides *YamlFileLoader*[22] there are two other loaders that work the same way:

- *XmlFileLoader*[23]
- *PhpFileLoader*[24]

If you use the *PhpFileLoader*[25] you have to provide the name of a PHP file which returns a *RouteCollection*[26]:

18. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
19. http://api.symfony.com/3.0/Symfony/Component/Config/FileLocator.html
20. http://api.symfony.com/3.0/Symfony/Component/Config/FileLocator.html
21. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html
22. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/YamlFileLoader.html
23. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/XmlFileLoader.html
24. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/PhpFileLoader.html
25. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/PhpFileLoader.html
26. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html

Listing 76-10

```
// RouteProvider.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add(
    'route_name',
    new Route('/foo', array('controller' => 'ExampleController'))
);
// ...

return $collection;
```

Routes as Closures

There is also the *ClosureLoader*[27], which calls a closure and uses the result as a *RouteCollection*[28]:

Listing 76-11

```
use Symfony\Component\Routing\Loader\ClosureLoader;

$closure = function () {
    return new RouteCollection();
};

$loader = new ClosureLoader();
$collection = $loader->load($closure);
```

Routes as Annotations

Last but not least there are *AnnotationDirectoryLoader*[29] and *AnnotationFileLoader*[30] to load route definitions from class annotations. The specific details are left out here.

The all-in-one Router

The *Router*[31] class is an all-in-one package to quickly use the Routing component. The constructor expects a loader instance, a path to the main route definition and some other settings:

Listing 76-12

```
public function __construct(
    LoaderInterface $loader,
    $resource,
    array $options = array(),
    RequestContext $context = null,
    array $defaults = array()
);
```

With the `cache_dir` option you can enable route caching (if you provide a path) or disable caching (if it's set to `null`). The caching is done automatically in the background if you want to use it. A basic example of the *Router*[32] class would look like:

Listing 76-13

27. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/ClosureLoader.html
28. http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html
29. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/AnnotationDirectoryLoader.html
30. http://api.symfony.com/3.0/Symfony/Component/Routing/Loader/AnnotationFileLoader.html
31. http://api.symfony.com/3.0/Symfony/Component/Routing/Router.html
32. http://api.symfony.com/3.0/Symfony/Component/Routing/Router.html

```php
$locator = new FileLocator(array(__DIR__));
$requestContext = new RequestContext('/');

$router = new Router(
    new YamlFileLoader($locator),
    'routes.yml',
    array('cache_dir' => __DIR__.'/cache'),
    $requestContext
);
$router->match('/foo/bar');
```

If you use caching, the Routing component will compile new classes which are saved in the `cache_dir`. This means your script must have write permissions for that location.

Chapter 77
How to Match a Route Based on the Host

You can also match on the HTTP *host* of the incoming request.

Listing 77-1
```
1 mobile_homepage:
2     path:     /
3     host:     m.example.com
4     defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }
5
6 homepage:
7     path:     /
8     defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Both routes match the same path `/`, however the first one will match only if the host is `m.example.com`.

Using Placeholders

The host option uses the same syntax as the path matching system. This means you can use placeholders in your hostname:

Listing 77-2
```
1 projects_homepage:
2     path:     /
3     host:     "{project_name}.example.com"
4     defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }
5
6 homepage:
7     path:     /
8     defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

You can also set requirements and default options for these placeholders. For instance, if you want to match both `m.example.com` and `mobile.example.com`, you use this:

Listing 77-3
```
1 mobile_homepage:
2     path:     /
```

```
3       host:     "{subdomain}.example.com"
4       defaults:
5           _controller: AcmeDemoBundle:Main:mobileHomepage
6           subdomain: m
7       requirements:
8           subdomain: m|mobile
9
10  homepage:
11      path:     /
12      defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

You can also use service parameters if you do not want to hardcode the hostname:

Listing 77-4

```
1  mobile_homepage:
2      path:     /
3      host:     "m.{domain}"
4      defaults:
5          _controller: AcmeDemoBundle:Main:mobileHomepage
6          domain: '%domain%'
7      requirements:
8          domain: '%domain%'
9
10 homepage:
11     path:  /
12     defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Make sure you also include a default option for the `domain` placeholder, otherwise you need to include a domain value each time you generate a URL using the route.

Using Host Matching of Imported Routes

You can also set the host option on imported routes:

Listing 77-5

```
1  acme_hello:
2      resource: '@AcmeHelloBundle/Resources/config/routing.yml'
3      host:     "hello.example.com"
```

The host `hello.example.com` will be set on each route loaded from the new routing resource.

Testing your Controllers

You need to set the Host HTTP header on your request objects if you want to get past url matching in your functional tests.

Listing 77-6

```
1  $crawler = $client->request(
2      'GET',
3      '/homepage',
4      array(),
```

```
5       array(),
6       array('HTTP_HOST' => 'm.' . $client->getContainer()->getParameter('domain'))
7   );
```

Chapter 78
The Security Component

The Security component provides a complete security system for your web application. It ships with facilities for authenticating using HTTP basic or digest authentication, interactive form login or X.509 certificate login, but also allows you to implement your own authentication strategies. Furthermore, the component provides ways to authorize authenticated users based on their roles, and it contains an advanced ACL system.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/security` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/security*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

The Security component is divided into four smaller sub-components which can be used separately:

`symfony/security-core`
: It provides all the common security features, from authentication to authorization and from encoding passwords to loading users.

`symfony/security-http`
: It integrates the core sub-component with the HTTP protocol to handle HTTP requests and responses.

`symfony/security-csrf`
: It provides protection against *CSRF attacks*[2].

`symfony/security-acl`
: It provides a fine grained permissions mechanism based on Access Control Lists.

1. https://packagist.org/packages/symfony/security
2. https://en.wikipedia.org/wiki/Cross-site_request_forgery

Sections

- *The Firewall and Authorization*
- *Authentication*
- *Authorization*
- *Securely Generating Random Values*

Chapter 79

The Firewall and Authorization

Central to the Security component is authorization. This is handled by an instance of *AuthorizationCheckerInterface*[1]. When all steps in the process of authenticating the user have been taken successfully, you can ask the authorization checker if the authenticated user has access to a certain action or resource of the application:

Listing 79-1

```
use Symfony\Component\Security\Core\Authorization\AuthorizationChecker;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// instance of
Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface
$tokenStorage = ...;

// instance of
Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface
$authenticationManager = ...;

// instance of Symfony\Component\Security\Core\Authorization\AccessDecisionManagerInterface
$accessDecisionManager = ...;

$authorizationChecker = new AuthorizationChecker(
    $tokenStorage,
    $authenticationManager,
    $accessDecisionManager
);

// ... authenticate the user

if (!$authorizationChecker->isGranted('ROLE_ADMIN')) {
    throw new AccessDeniedException();
}
```

1. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AuthorizationCheckerInterface.html

Read the dedicated sections to learn more about *Authentication* and *Authorization*.

A Firewall for HTTP Requests

Authenticating a user is done by the firewall. An application may have multiple secured areas, so the firewall is configured using a map of these secured areas. For each of these areas, the map contains a request matcher and a collection of listeners. The request matcher gives the firewall the ability to find out if the current request points to a secured area. The listeners are then asked if the current request can be used to authenticate the user:

Listing 79-2

```
use Symfony\Component\Security\Http\FirewallMap;
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\ExceptionListener;

$map = new FirewallMap();

$requestMatcher = new RequestMatcher('^/secured-area/');

// instances of Symfony\Component\Security\Http\Firewall\ListenerInterface
$listeners = array(...);

$exceptionListener = new ExceptionListener(...);

$map->add($requestMatcher, $listeners, $exceptionListener);
```

The firewall map will be given to the firewall as its first argument, together with the event dispatcher that is used by the `HttpKernel`[2]:

Listing 79-3

```
use Symfony\Component\Security\Http\Firewall;
use Symfony\Component\HttpKernel\KernelEvents;

// the EventDispatcher used by the HttpKernel
$dispatcher = ...;

$firewall = new Firewall($map, $dispatcher);

$dispatcher->addListener(
    KernelEvents::REQUEST,
    array($firewall, 'onKernelRequest')
);
```

The firewall is registered to listen to the `kernel.request` event that will be dispatched by the HttpKernel at the beginning of each request it processes. This way, the firewall may prevent the user from going any further than allowed.

Firewall Listeners

When the firewall gets notified of the `kernel.request` event, it asks the firewall map if the request matches one of the secured areas. The first secured area that matches the request will return a set of corresponding firewall listeners (which each implement `ListenerInterface`[3]). These listeners will all

2. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html

be asked to handle the current request. This basically means: find out if the current request contains any information by which the user might be authenticated (for instance the Basic HTTP authentication listener checks if the request has a header called `PHP_AUTH_USER`).

Exception Listener

If any of the listeners throws an *AuthenticationException*[4], the exception listener that was provided when adding secured areas to the firewall map will jump in.

The exception listener determines what happens next, based on the arguments it received when it was created. It may start the authentication procedure, perhaps ask the user to supply their credentials again (when they have only been authenticated based on a "remember-me" cookie), or transform the exception into an *AccessDeniedHttpException*[5], which will eventually result in an "HTTP/1.1 403: Access Denied" response.

Entry Points

When the user is not authenticated at all (i.e. when the token storage has no token yet), the firewall's entry point will be called to "start" the authentication process. An entry point should implement *AuthenticationEntryPointInterface*[6], which has only one method: *start()*[7]. This method receives the current *Request*[8] object and the exception by which the exception listener was triggered. The method should return a *Response*[9] object. This could be, for instance, the page containing the login form or, in the case of Basic HTTP authentication, a response with a `WWW-Authenticate` header, which will prompt the user to supply their username and password.

Flow: Firewall, Authentication, Authorization

Hopefully you can now see a little bit about how the "flow" of the security context works:

1. The Firewall is registered as a listener on the `kernel.request` event;
2. At the beginning of the request, the Firewall checks the firewall map to see if any firewall should be active for this URL;
3. If a firewall is found in the map for this URL, its listeners are notified;
4. Each listener checks to see if the current request contains any authentication information - a listener may (a) authenticate a user, (b) throw an `AuthenticationException`, or (c) do nothing (because there is no authentication information on the request);
5. Once a user is authenticated, you'll use *Authorization* to deny access to certain resources.

Read the next sections to find out more about *Authentication* and *Authorization*.

3. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/ListenerInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html
5. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Exception/AccessDeniedHttpException.html
6. http://api.symfony.com/3.0/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html
7. http://api.symfony.com/3.0/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html#method_start
8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
9. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html

Chapter 80

Authentication

When a request points to a secured area, and one of the listeners from the firewall map is able to extract the user's credentials from the current `Request`[1] object, it should create a token, containing these credentials. The next thing the listener should do is ask the authentication manager to validate the given token, and return an *authenticated* token if the supplied credentials were found to be valid. The listener should then store the authenticated token using `the token storage`[2]:

Listing 80-1

```
use Symfony\Component\Security\Http\Firewall\ListenerInterface;
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;
use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;

class SomeAuthenticationListener implements ListenerInterface
{
    /**
     * @var TokenStorageInterface
     */
    private $tokenStorage;

    /**
     * @var AuthenticationManagerInterface
     */
    private $authenticationManager;

    /**
     * @var string Uniquely identifies the secured area
     */
    private $providerKey;

    // ...

    public function handle(GetResponseEvent $event)
    {
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
2. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/Storage/TokenStorageInterface.html

```
28         $request = $event->getRequest();
29
30         $username = ...;
31         $password = ...;
32
33         $unauthenticatedToken = new UsernamePasswordToken(
34             $username,
35             $password,
36             $this->providerKey
37         );
38
39         $authenticatedToken = $this
40             ->authenticationManager
41             ->authenticate($unauthenticatedToken);
42
43         $this->tokenStorage->setToken($authenticatedToken);
44     }
45 }
```

A token can be of any class, as long as it implements *TokenInterface*[3].

The Authentication Manager

The default authentication manager is an instance of *AuthenticationProviderManager*[4]:

Listing 80-2

```
1  use Symfony\Component\Security\Core\Authentication\AuthenticationProviderManager;
2
3  // instances of
4  Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterface
5  $providers = array(...);
6
7  $authenticationManager = new AuthenticationProviderManager($providers);
8
9  try {
10     $authenticatedToken = $authenticationManager
11         ->authenticate($unauthenticatedToken);
12 } catch (AuthenticationException $failed) {
13     // authentication failed
   }
```

The `AuthenticationProviderManager`, when instantiated, receives several authentication providers, each supporting a different type of token.

You may of course write your own authentication manager, it only has to implement *AuthenticationManagerInterface*[5].

3. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/AuthenticationProviderManager.html
5. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/AuthenticationManagerInterface.html

Authentication Providers

Each provider (since it implements *AuthenticationProviderInterface*[6]) has a method *supports()*[7] by which the `AuthenticationProviderManager` can determine if it supports the given token. If this is the case, the manager then calls the provider's method *authenticate()*[8]. This method should return an authenticated token or throw an *AuthenticationException*[9] (or any other exception extending it).

Authenticating Users by their Username and Password

An authentication provider will attempt to authenticate a user based on the credentials they provided. Usually these are a username and a password. Most web applications store their user's username and a hash of the user's password combined with a randomly generated salt. This means that the average authentication would consist of fetching the salt and the hashed password from the user data storage, hash the password the user has just provided (e.g. using a login form) with the salt and compare both to determine if the given password is valid.

This functionality is offered by the *DaoAuthenticationProvider*[10]. It fetches the user's data from a *UserProviderInterface*[11], uses a *PasswordEncoderInterface*[12] to create a hash of the password and returns an authenticated token if the password was valid:

Listing 80-3

```
use Symfony\Component\Security\Core\Authentication\Provider\DaoAuthenticationProvider;
use Symfony\Component\Security\Core\User\UserChecker;
use Symfony\Component\Security\Core\User\InMemoryUserProvider;
use Symfony\Component\Security\Core\Encoder\EncoderFactory;

$userProvider = new InMemoryUserProvider(
    array(
        'admin' => array(
            // password is "foo"
            'password' =>
'5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg==',
            'roles'    => array('ROLE_ADMIN'),
        ),
    )
);

// for some extra checks: is account enabled, locked, expired, etc.?
$userChecker = new UserChecker();

// an array of password encoders (see below)
$encoderFactory = new EncoderFactory(...);

$provider = new DaoAuthenticationProvider(
    $userProvider,
    $userChecker,
    'secured_area',
    $encoderFactory
);
```

6. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html
7. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html#method_supports
8. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html#method_authenticate
9. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html
10. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
11. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
12. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
28
29 $provider->authenticate($unauthenticatedToken);
```

The example above demonstrates the use of the "in-memory" user provider, but you may use any user provider, as long as it implements *UserProviderInterface*[13]. It is also possible to let multiple user providers try to find the user's data, using the *ChainUserProvider*[14].

The Password Encoder Factory

The *DaoAuthenticationProvider*[15] uses an encoder factory to create a password encoder for a given type of user. This allows you to use different encoding strategies for different types of users. The default *EncoderFactory*[16] receives an array of encoders:

Listing 80-4

```
1  use Symfony\Component\Security\Core\Encoder\EncoderFactory;
2  use Symfony\Component\Security\Core\Encoder\MessageDigestPasswordEncoder;
3
4  $defaultEncoder = new MessageDigestPasswordEncoder('sha512', true, 5000);
5  $weakEncoder = new MessageDigestPasswordEncoder('md5', true, 1);
6
7  $encoders = array(
8      'Symfony\\Component\\Security\\Core\\User\\User' => $defaultEncoder,
9      'Acme\\Entity\\LegacyUser'                       => $weakEncoder,
10
11     // ...
12 );
13
14 $encoderFactory = new EncoderFactory($encoders);
```

Each encoder should implement *PasswordEncoderInterface*[17] or be an array with a `class` and an `arguments` key, which allows the encoder factory to construct the encoder only when it is needed.

Creating a custom Password Encoder

There are many built-in password encoders. But if you need to create your own, it just needs to follow these rules:

1. The class must implement *PasswordEncoderInterface*[18];
2. The implementations of *encodePassword()*[19] and *isPasswordValid()*[20] must first of all make sure the password is not too long, i.e. the password length is no longer than 4096 characters. This is for security reasons (see *CVE-2013-5750*[21]), and you can use the *isPasswordTooLong()*[22] method for this check:

Listing 80-5

13. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
14. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/ChainUserProvider.html
15. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
16. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/EncoderFactory.html
17. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html
18. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html
19. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html#method_encodePassword
20. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html#method_isPasswordValid
21. https://symfony.com/blog/cve-2013-5750-security-issue-in-fosuserbundle-login-form
22. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/BasePasswordEncoder.html#method_isPasswordTooLong

```php
use Symfony\Component\Security\Core\Encoder\BasePasswordEncoder;
use Symfony\Component\Security\Core\Exception\BadCredentialsException;

class FoobarEncoder extends BasePasswordEncoder
{
    public function encodePassword($raw, $salt)
    {
        if ($this->isPasswordTooLong($raw)) {
            throw new BadCredentialsException('Invalid password.');
        }

        // ...
    }

    public function isPasswordValid($encoded, $raw, $salt)
    {
        if ($this->isPasswordTooLong($raw)) {
            return false;
        }

        // ...
}
```

Using Password Encoders

When the *getEncoder()*[23] method of the password encoder factory is called with the user object as its first argument, it will return an encoder of type *PasswordEncoderInterface*[24] which should be used to encode this user's password:

Listing 80-6

```php
// a Acme\Entity\LegacyUser instance
$user = ...;

// the password that was submitted, e.g. when registering
$plainPassword = ...;

$encoder = $encoderFactory->getEncoder($user);

// will return $weakEncoder (see above)
$encodedPassword = $encoder->encodePassword($plainPassword, $user->getSalt());

$user->setPassword($encodedPassword);

// ... save the user
```

Now, when you want to check if the submitted password (e.g. when trying to log in) is correct, you can use:

Listing 80-7

```php
// fetch the Acme\Entity\LegacyUser
$user = ...;

// the submitted password, e.g. from the login form
$plainPassword = ...;

```

23. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/EncoderFactory.html#method_getEncoder
24. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
$validPassword = $encoder->isPasswordValid(
    $user->getPassword(), // the encoded password
    $plainPassword,       // the submitted password
    $user->getSalt()
);
```

Authentication Events

The security component provides 4 related authentication events:

Name	Event Constant	Argument Passed to the Listener
security.authentication.success	`AuthenticationEvents::AUTHENTICATION_SUCCESS`	*AuthenticationEvent*[25]
security.authentication.failure	`AuthenticationEvents::AUTHENTICATION_FAILURE`	*AuthenticationFailureEven*
security.interactive_login	`SecurityEvents::INTERACTIVE_LOGIN`	*InteractiveLoginEvent*[27]
security.switch_user	`SecurityEvents::SWITCH_USER`	*SwitchUserEvent*[28]

Authentication Success and Failure Events

When a provider authenticates the user, a `security.authentication.success` event is dispatched. But beware - this event will fire, for example, on *every* request if you have session-based authentication. See `security.interactive_login` below if you need to do something when a user *actually* logs in.

When a provider attempts authentication but fails (i.e. throws an `AuthenticationException`), a `security.authentication.failure` event is dispatched. You could listen on the `security.authentication.failure` event, for example, in order to log failed login attempts.

Security Events

The `security.interactive_login` event is triggered after a user has actively logged into your website. It is important to distinguish this action from non-interactive authentication methods, such as:

- authentication based on a "remember me" cookie.
- authentication based on your session.
- authentication using a HTTP basic or HTTP digest header.

You could listen on the `security.interactive_login` event, for example, in order to give your user a welcome flash message every time they log in.

The `security.switch_user` event is triggered every time you activate the `switch_user` firewall listener.

For more information on switching users, see How to Impersonate a User.

25. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Event/AuthenticationEvent.html
26. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Event/AuthenticationFailureEvent.html
27. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Event/InteractiveLoginEvent.html
28. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Event/SwitchUserEvent.html

Chapter 81
Authorization

When any of the authentication providers (see Authentication Providers) has verified the still-unauthenticated token, an authenticated token will be returned. The authentication listener should set this token directly in the *TokenStorageInterface*[1] using its *setToken()*[2] method.

From then on, the user is authenticated, i.e. identified. Now, other parts of the application can use the token to decide whether or not the user may request a certain URI, or modify a certain object. This decision will be made by an instance of *AccessDecisionManagerInterface*[3].

An authorization decision will always be based on a few things:

- **The current token**

  For instance, the token's *getRoles()*[4] method may be used to retrieve the roles of the current user (e.g. `ROLE_SUPER_ADMIN`), or a decision may be based on the class of the token.

- **A set of attributes**

  Each attribute stands for a certain right the user should have, e.g. `ROLE_ADMIN` to make sure the user is an administrator.

- **An object (optional)**

  Any object for which access control needs to be checked, like an article or a comment object.

Access Decision Manager

Since deciding whether or not a user is authorized to perform a certain action can be a complicated process, the standard *AccessDecisionManager*[5] itself depends on multiple voters, and makes a final

1. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/Storage/TokenStorageInterface.html
2. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/Storage/TokenStorageInterface.html#method_setToken
3. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AccessDecisionManagerInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#method_getRoles
5. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AccessDecisionManager.html

verdict based on all the votes (either positive, negative or neutral) it has received. It recognizes several strategies:

`affirmative` **(default)**
: grant access as soon as there is one voter granting access;

`consensus`
: grant access if there are more voters granting access than there are denying;

`unanimous`
: only grant access if none of the voters has denied access;

Listing 81-1

```
use Symfony\Component\Security\Core\Authorization\AccessDecisionManager;

// instances of Symfony\Component\Security\Core\Authorization\Voter\VoterInterface
$voters = array(...);

// one of "affirmative", "consensus", "unanimous"
$strategy = ...;

// whether or not to grant access when all voters abstain
$allowIfAllAbstainDecisions = ...;

// whether or not to grant access when there is no majority (applies only to the
"consensus" strategy)
$allowIfEqualGrantedDeniedDecisions = ...;

$accessDecisionManager = new AccessDecisionManager(
    $voters,
    $strategy,
    $allowIfAllAbstainDecisions,
    $allowIfEqualGrantedDeniedDecisions
);
```

You can change the default strategy in the configuration.

Voters

Voters are instances of *VoterInterface*[6], which means they have to implement a few methods which allows the decision manager to use them:

`vote(TokenInterface $token, $object, array $attributes)`
: this method will do the actual voting and return a value equal to one of the class constants of *VoterInterface*[7], i.e. `VoterInterface::ACCESS_GRANTED`, `VoterInterface::ACCESS_DENIED` or `VoterInterface::ACCESS_ABSTAIN`;

The Security component contains some standard voters which cover many use cases:

AuthenticatedVoter

The *AuthenticatedVoter*[8] voter supports the attributes `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY` and grants access based on the

6. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html
7. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html
8. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/AuthenticatedVoter.html

current level of authentication, i.e. is the user fully authenticated, or only based on a "remember-me" cookie, or even authenticated anonymously?

Listing 81-2

```
use Symfony\Component\Security\Core\Authentication\AuthenticationTrustResolver;

$anonymousClass = 'Symfony\Component\Security\Core\Authentication\Token\AnonymousToken';
$rememberMeClass = 'Symfony\Component\Security\Core\Authentication\Token\RememberMeToken';

$trustResolver = new AuthenticationTrustResolver($anonymousClass, $rememberMeClass);

$authenticatedVoter = new AuthenticatedVoter($trustResolver);

// instance of Symfony\Component\Security\Core\Authentication\Token\TokenInterface
$token = ...;

// any object
$object = ...;

$vote = $authenticatedVoter->vote($token, $object, array('IS_AUTHENTICATED_FULLY'));
```

RoleVoter

The *RoleVoter*[9] supports attributes starting with ROLE_ and grants access to the user when the required ROLE_* attributes can all be found in the array of roles returned by the token's *getRoles()*[10] method:

Listing 81-3

```
use Symfony\Component\Security\Core\Authorization\Voter\RoleVoter;

$roleVoter = new RoleVoter('ROLE_');

$roleVoter->vote($token, $object, array('ROLE_ADMIN'));
```

RoleHierarchyVoter

The *RoleHierarchyVoter*[11] extends *RoleVoter*[12] and provides some additional functionality: it knows how to handle a hierarchy of roles. For instance, a ROLE_SUPER_ADMIN role may have subroles ROLE_ADMIN and ROLE_USER, so that when a certain object requires the user to have the ROLE_ADMIN role, it grants access to users who in fact have the ROLE_ADMIN role, but also to users having the ROLE_SUPER_ADMIN role:

Listing 81-4

```
use Symfony\Component\Security\Core\Authorization\Voter\RoleHierarchyVoter;
use Symfony\Component\Security\Core\Role\RoleHierarchy;

$hierarchy = array(
    'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_USER'),
);

$roleHierarchy = new RoleHierarchy($hierarchy);

$roleHierarchyVoter = new RoleHierarchyVoter($roleHierarchy);
```

9. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html
10. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#method_getRoles
11. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/RoleHierarchyVoter.html
12. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html

When you make your own voter, you may of course use its constructor to inject any dependencies it needs to come to a decision.

Roles

Roles are objects that give expression to a certain right the user has. The only requirement is that they implement *RoleInterface*[13], which means they should also have a *getRole()*[14] method that returns a string representation of the role itself. The default *Role*[15] simply returns its first constructor argument:

Listing 81-5

```
use Symfony\Component\Security\Core\Role\Role;

$role = new Role('ROLE_ADMIN');

// will show 'ROLE_ADMIN'
var_dump($role->getRole());
```

Most authentication tokens extend from *AbstractToken*[16], which means that the roles given to its constructor will be automatically converted from strings to these simple `Role` objects.

Using the Decision Manager

The Access Listener

The access decision manager can be used at any point in a request to decide whether or not the current user is entitled to access a given resource. One optional, but useful, method for restricting access based on a URL pattern is the *AccessListener*[17], which is one of the firewall listeners (see Firewall Listeners) that is triggered for each request matching the firewall map (see A Firewall for HTTP Requests).

It uses an access map (which should be an instance of *AccessMapInterface*[18]) which contains request matchers and a corresponding set of attributes that are required for the current user to get access to the application:

Listing 81-6

```
use Symfony\Component\Security\Http\AccessMap;
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\AccessListener;

$accessMap = new AccessMap();
$requestMatcher = new RequestMatcher('^/admin');
$accessMap->add($requestMatcher, array('ROLE_ADMIN'));

$accessListener = new AccessListener(
    $securityContext,
```

13. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Role/RoleInterface.html
14. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Role/RoleInterface.html#method_getRole
15. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Role/Role.html
16. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/AbstractToken.html
17. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/AccessListener.html
18. http://api.symfony.com/3.0/Symfony/Component/Security/Http/AccessMapInterface.html

```
11     $accessDecisionManager,
12     $accessMap,
13     $authenticationManager
14 );
```

Authorization Checker

The access decision manager is also available to other parts of the application via the *isGranted()*[19] method of the *AuthorizationChecker*[20]. A call to this method will directly delegate the question to the access decision manager:

Listing 81-7

```
1  use Symfony\Component\Security\Core\Authorization\AuthorizationChecker;
2  use Symfony\Component\Security\Core\Exception\AccessDeniedException;
3
4  $authorizationChecker = new AuthorizationChecker(
5      $tokenStorage,
6      $authenticationManager,
7      $accessDecisionManager
8  );
9
10 if (!$authorizationChecker->isGranted('ROLE_ADMIN')) {
11     throw new AccessDeniedException();
12 }
```

19. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AuthorizationChecker.html#method_isGranted
20. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AuthorizationChecker.html

Chapter 82

Securely Generating Random Values

The Symfony Security component comes with a collection of nice utilities related to security. These utilities are used by Symfony, but you should also use them if you want to solve the problem they address.

The functions described in this article were introduced in PHP 5.6 or 7. For older PHP versions, a polyfill is provided by the *Symfony Polyfill Component*[1].

Comparing Strings

The time it takes to compare two strings depends on their differences. This can be used by an attacker when the two strings represent a password for instance; it is known as a *Timing attack*[2].

When comparing two passwords, you should use the *hash_equals*[3] function:

Listing 82-1
```
if (hash_equals($knownString, $userInput)) {
    // ...
}
```

Generating a Secure Random String

Whenever you need to generate a secure random string, you are highly encouraged to use the *random_bytes*[4] function:

Listing 82-2
```
$random = random_bytes(10);
```

The function returns a random string, suitable for cryptographic use, of the number bytes passed as an argument (10 in the above example).

1. https://github.com/symfony/polyfill
2. https://en.wikipedia.org/wiki/Timing_attack
3. http://php.net/manual/en/function.hash-equals.php
4. http://php.net/manual/en/function.random-bytes.php

The `random_bytes()` function returns a binary string which may contain the `\0` character. This can cause trouble in several common scenarios, such as storing this value in a database or including it as part of the URL. The solution is to encode or hash the value returned by `random_bytes()` (to do that, you can use a simple `base64_encode()` PHP function).

Generating a Secure Random Number

If you need to generate a cryptographically secure random integer, you should use the *random_int*[5] function:

Listing 82-3
```
$random = random_int(1, 10);
```

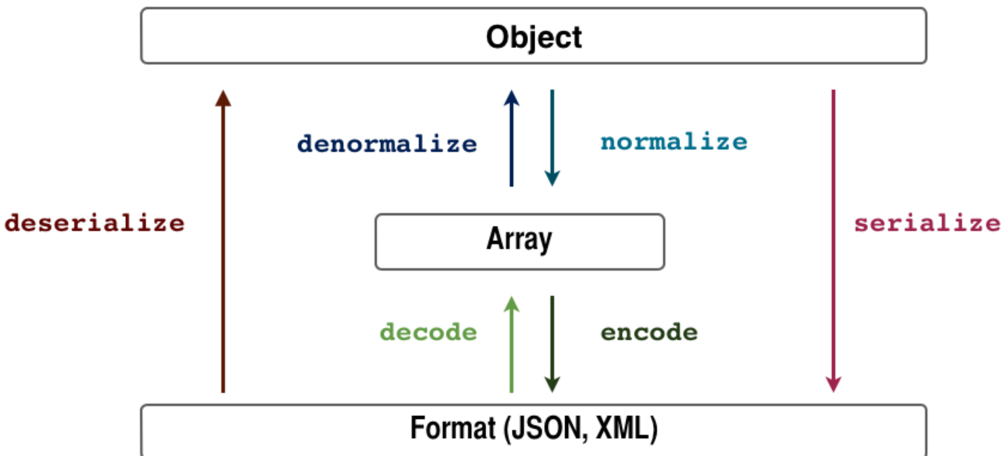
5. http://php.net/manual/en/function.random-int.php

Chapter 83

The Serializer Component

> The Serializer component is meant to be used to turn objects into a specific format (XML, JSON, YAML, ...) and the other way around.

In order to do so, the Serializer component follows the following simple schema.



As you can see in the picture above, an array is used as a man in the middle. This way, Encoders will only deal with turning specific **formats** into **arrays** and vice versa. The same way, Normalizers will deal with turning specific **objects** into **arrays** and vice versa.

Serialization is a complicated topic, and while this component may not work in all cases, it can be a useful tool while developing tools to serialize and deserialize your objects.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/serializer` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/serializer*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

To use the `ObjectNormalizer`, the *PropertyAccess component* must also be installed.

Usage

Using the Serializer component is really simple. You just need to set up the *`Serializer`*[2] specifying which Encoders and Normalizer are going to be available:

Listing 83-1

```
1  use Symfony\Component\Serializer\Serializer;
2  use Symfony\Component\Serializer\Encoder\XmlEncoder;
3  use Symfony\Component\Serializer\Encoder\JsonEncoder;
4  use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
5
6  $encoders = array(new XmlEncoder(), new JsonEncoder());
7  $normalizers = array(new ObjectNormalizer());
8
9  $serializer = new Serializer($normalizers, $encoders);
```

The preferred normalizer is the *`ObjectNormalizer`*[3], but other normalizers are available. To read more about them, refer to the Normalizers section of this page. All the examples shown below use the `ObjectNormalizer`.

Serializing an Object

For the sake of this example, assume the following class already exists in your project:

Listing 83-2

```
1   namespace Acme;
2
3   class Person
4   {
5       private $age;
6       private $name;
7       private $sportsman;
8
9       // Getters
10      public function getName()
11      {
12          return $this->name;
13      }
14
15      public function getAge()
16      {
17          return $this->age;
18      }
19
20      // Issers
```

1. https://packagist.org/packages/symfony/serializer
2. http://api.symfony.com/3.0/Symfony/Component/Serializer/Serializer.html
3. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/ObjectNormalizer.html

```
21     public function isSportsman()
22     {
23         return $this->sportsman;
24     }
25
26     // Setters
27     public function setName($name)
28     {
29         $this->name = $name;
30     }
31
32     public function setAge($age)
33     {
34         $this->age = $age;
35     }
36
37     public function setSportsman($sportsman)
38     {
39         $this->sportsman = $sportsman;
40     }
41 }
```

Now, if you want to serialize this object into JSON, you only need to use the Serializer service created before:

Listing 83-3
```
1  $person = new Acme\Person();
2  $person->setName('foo');
3  $person->setAge(99);
4  $person->setSportsman(false);
5
6  $jsonContent = $serializer->serialize($person, 'json');
7
8  // $jsonContent contains {"name":"foo","age":99,"sportsman":false}
9
10 echo $jsonContent; // or return it in a Response
```

The first parameter of the *serialize()*[4] is the object to be serialized and the second is used to choose the proper encoder, in this case *JsonEncoder*[5].

Deserializing an Object

You'll now learn how to do the exact opposite. This time, the information of the `Person` class would be encoded in XML format:

Listing 83-4
```
1 $data = <<<EOF
2 <person>
3     <name>foo</name>
4     <age>99</age>
5     <sportsman>false</sportsman>
6 </person>
7 EOF;
```

4. http://api.symfony.com/3.0/Symfony/Component/Serializer/Serializer.html#method_serialize
5. http://api.symfony.com/3.0/Symfony/Component/Serializer/Encoder/JsonEncoder.html

```
8
9 $person = $serializer->deserialize($data, 'Acme\Person', 'xml');
```

In this case, *deserialize()*[6] needs three parameters:

1. The information to be decoded
2. The name of the class this information will be decoded to
3. The encoder used to convert that information into an array

Deserializing in an Existing Object

The serializer can also be used to update an existing object:

Listing 83-5

```
1  $person = new Acme\Person();
2  $person->setName('bar');
3  $person->setAge(99);
4  $person->setSportsman(true);
5
6  $data = <<<EOF
7  <person>
8      <name>foo</name>
9      <age>69</age>
10 </person>
11 EOF;
12
13 $serializer->deserialize($data, 'Acme\Person', 'xml', array('object_to_populate' =>
14 $person));
   // $obj2 = Acme\Person(name: 'foo', age: '99', sportsman: true)
```

This is a common need when working with an ORM.

Attributes Groups

Sometimes, you want to serialize different sets of attributes from your entities. Groups are a handy way to achieve this need.

Assume you have the following plain-old-PHP object:

Listing 83-6

```
1  namespace Acme;
2
3  class MyObj
4  {
5      public $foo;
6
7      private $bar;
8
9      public function getBar()
10     {
11         return $this->bar;
12     }
13
14     public function setBar($bar)
15     {
16         return $this->bar = $bar;
```

6. http://api.symfony.com/3.0/Symfony/Component/Serializer/Serializer.html#method_deserialize

```
17      }
18  }
```

The definition of serialization can be specified using annotations, XML or YAML. The *ClassMetadataFactory*[7] that will be used by the normalizer must be aware of the format to use.

Initialize the *ClassMetadataFactory*[8] like the following:

Listing 83-7

```
1  use Symfony\Component\Serializer\Mapping\Factory\ClassMetadataFactory;
2  // For annotations
3  use Doctrine\Common\Annotations\AnnotationReader;
4  use Symfony\Component\Serializer\Mapping\Loader\AnnotationLoader;
5  // For XML
6  // use Symfony\Component\Serializer\Mapping\Loader\XmlFileLoader;
7  // For YAML
8  // use Symfony\Component\Serializer\Mapping\Loader\YamlFileLoader;
9
10 $classMetadataFactory = new ClassMetadataFactory(new AnnotationLoader(new
11 AnnotationReader()));
12 // For XML
13 // $classMetadataFactory = new ClassMetadataFactory(new XmlFileLoader('/path/to/your/
14 definition.xml'));
   // For YAML
   // $classMetadataFactory = new ClassMetadataFactory(new YamlFileLoader('/path/to/your/
   definition.yml'));
```

Then, create your groups definition:

Listing 83-8

```
1  namespace Acme;
2
3  use Symfony\Component\Serializer\Annotation\Groups;
4
5  class MyObj
6  {
7      /**
8       * @Groups({"group1", "group2"})
9       */
10     public $foo;
11
12     /**
13      * @Groups({"group3"})
14      */
15     public function getBar() // is* methods are also supported
16     {
17         return $this->bar;
18     }
19
20     // ...
21 }
```

You are now able to serialize only attributes in the groups you want:

Listing 83-9

```
1  use Symfony\Component\Serializer\Serializer;
2  use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
```

7. http://api.symfony.com/3.0/Symfony/Component/Serializer/Mapping/Factory/ClassMetadataFactory.html
8. http://api.symfony.com/3.0/Symfony/Component/Serializer/Mapping/Factory/ClassMetadataFactory.html

```
3
4 $obj = new MyObj();
5 $obj->foo = 'foo';
6 $obj->setBar('bar');
7
8 $normalizer = new ObjectNormalizer($classMetadataFactory);
9 $serializer = new Serializer(array($normalizer));
10
11 $data = $serializer->normalize($obj, null, array('groups' => array('group1')));
12 // $data = array('foo' => 'foo');
13
14 $obj2 = $serializer->denormalize(
15     array('foo' => 'foo', 'bar' => 'bar'),
16     'MyObj',
17     null,
18     array('groups' => array('group1', 'group3'))
19 );
20 // $obj2 = MyObj(foo: 'foo', bar: 'bar')
```

Ignoring Attributes

Using attribute groups instead of the *setIgnoredAttributes()*[9] method is considered best practice.

As an option, there's a way to ignore attributes from the origin object. To remove those attributes use the *setIgnoredAttributes()*[10] method on the normalizer definition:

Listing 83-10

```
1 use Symfony\Component\Serializer\Serializer;
2 use Symfony\Component\Serializer\Encoder\JsonEncoder;
3 use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
4
5 $normalizer = new ObjectNormalizer();
6 $normalizer->setIgnoredAttributes(array('age'));
7 $encoder = new JsonEncoder();
8
9 $serializer = new Serializer(array($normalizer), array($encoder));
10 $serializer->serialize($person, 'json'); // Output: {"name":"foo","sportsman":false}
```

Converting Property Names when Serializing and Deserializing

Sometimes serialized attributes must be named differently than properties or getter/setter methods of PHP classes.

The Serializer Component provides a handy way to translate or map PHP field names to serialized names: The Name Converter System.

Given you have the following object:

Listing 83-11

9. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/AbstractNormalizer.html#method_setIgnoredAttributes
10. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/AbstractNormalizer.html#method_setIgnoredAttributes

```
1 class Company
2 {
3     public $name;
4     public $address;
5 }
```

And in the serialized form, all attributes must be prefixed by `org_` like the following:

Listing 83-12
```
{"org_name": "Acme Inc.", "org_address": "123 Main Street, Big City"}
```

A custom name converter can handle such cases:

Listing 83-13
```
1  use Symfony\Component\Serializer\NameConverter\NameConverterInterface;
2
3  class OrgPrefixNameConverter implements NameConverterInterface
4  {
5      public function normalize($propertyName)
6      {
7          return 'org_'.$propertyName;
8      }
9
10     public function denormalize($propertyName)
11     {
12         // remove org_ prefix
13         return 'org_' === substr($propertyName, 0, 4) ? substr($propertyName, 4) :
14 $propertyName;
15     }
   }
```

The custom normalizer can be used by passing it as second parameter of any class extending *`AbstractNormalizer`*[11], including *`GetSetMethodNormalizer`*[12] and *`PropertyNormalizer`*[13]:

Listing 83-14
```
1  use Symfony\Component\Serializer\Encoder\JsonEncoder
2  use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
3  use Symfony\Component\Serializer\Serializer;
4
5  $nameConverter = new OrgPrefixNameConverter();
6  $normalizer = new ObjectNormalizer(null, $nameConverter);
7
8  $serializer = new Serializer(array($normalizer), array(new JsonEncoder()));
9
10 $obj = new Company();
11 $obj->name = 'Acme Inc.';
12 $obj->address = '123 Main Street, Big City';
13
14 $json = $serializer->serialize($obj);
15 // {"org_name": "Acme Inc.", "org_address": "123 Main Street, Big City"}
16 $objCopy = $serializer->deserialize($json);
17 // Same data as $obj
```

11. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/AbstractNormalizer.html
12. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html
13. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/PropertyNormalizer.html

CamelCase to snake_case

In many formats, it's common to use underscores to separate words (also known as snake_case). However, PSR-1 specifies that the preferred style for PHP properties and methods is CamelCase.

Symfony provides a built-in name converter designed to transform between snake_case and CamelCased styles during serialization and deserialization processes:

Listing 83-15

```
use Symfony\Component\Serializer\NameConverter\CamelCaseToSnakeCaseNameConverter;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;

$normalizer = new ObjectNormalizer(null, new CamelCaseToSnakeCaseNameConverter());

class Person
{
    private $firstName;

    public function __construct($firstName)
    {
        $this->firstName = $firstName;
    }

    public function getFirstName()
    {
        return $this->firstName;
    }
}

$kevin = new Person('Kévin');
$normalizer->normalize($kevin);
// ['first_name' => 'Kévin'];

$anne = $normalizer->denormalize(array('first_name' => 'Anne'), 'Person');
// Person object with firstName: 'Anne'
```

Serializing Boolean Attributes

If you are using isser methods (methods prefixed by `is`, like `Acme\Person::isSportsman()`), the Serializer component will automatically detect and use it to serialize related attributes.

The `ObjectNormalizer` also takes care of methods starting with `has`, `add` and `remove`.

Using Callbacks to Serialize Properties with Object Instances

When serializing, you can set a callback to format a specific object property:

Listing 83-16

```
use Acme\Person;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;
use Symfony\Component\Serializer\Serializer;

$encoder = new JsonEncoder();
$normalizer = new GetSetMethodNormalizer();

$callback = function ($dateTime) {
```

```
10      return $dateTime instanceof \DateTime
11          ? $dateTime->format(\DateTime::ISO8601)
12          : '';
13  };
14
15  $normalizer->setCallbacks(array('createdAt' => $callback));
16
17  $serializer = new Serializer(array($normalizer), array($encoder));
18
19  $person = new Person();
20  $person->setName('cordoval');
21  $person->setAge(34);
22  $person->setCreatedAt(new \DateTime('now'));
23
24  $serializer->serialize($person, 'json');
25  // Output: {"name":"cordoval", "age": 34, "createdAt": "2014-03-22T09:43:12-0500"}
```

Normalizers

There are several types of normalizers available:

`ObjectNormalizer`[14]

This normalizer leverages the *PropertyAccess Component* to read and write in the object. It means that it can access to properties directly and through getters, setters, hassers, adders and removers. It supports calling the constructor during the denormalization process.

Objects are normalized to a map of property names (method name stripped of the "get"/"set"/"has"/"remove" prefix and converted to lower case) to property values.

The `ObjectNormalizer` is the most powerful normalizer. It is a configured by default when using the Symfony Standard Edition with the serializer enabled.

`GetSetMethodNormalizer`[15]

This normalizer reads the content of the class by calling the "getters" (public methods starting with "get"). It will denormalize data by calling the constructor and the "setters" (public methods starting with "set").

Objects are normalized to a map of property names (method name stripped of the "get" prefix and converted to lower case) to property values.

`PropertyNormalizer`[16]

This normalizer directly reads and writes public properties as well as **private and protected** properties. It supports calling the constructor during the denormalization process.

Objects are normalized to a map of property names to property values.

Handling Circular References

Circular references are common when dealing with entity relations:

Listing 83-17

14. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/ObjectNormalizer.html
15. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html
16. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/PropertyNormalizer.html

```php
class Organization
{
    private $name;
    private $members;

    public function setName($name)
    {
        $this->name = $name;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setMembers(array $members)
    {
        $this->members = $members;
    }

    public function getMembers()
    {
        return $this->members;
    }
}

class Member
{
    private $name;
    private $organization;

    public function setName($name)
    {
        $this->name = $name;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setOrganization(Organization $organization)
    {
        $this->organization = $organization;
    }

    public function getOrganization()
    {
        return $this->organization;
    }
}
```

To avoid infinite loops, *GetSetMethodNormalizer*[17] throws a *CircularReferenceException*[18] when such a case is encountered:

Listing 83-18

17. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html
18. http://api.symfony.com/3.0/Symfony/Component/Serializer/Exception/CircularReferenceException.html

```php
1  $member = new Member();
2  $member->setName('Kévin');
3
4  $org = new Organization();
5  $org->setName('Les-Tilleuls.coop');
6  $org->setMembers(array($member));
7
8  $member->setOrganization($org);
9
10 echo $serializer->serialize($org, 'json'); // Throws a CircularReferenceException
```

The `setCircularReferenceLimit()` method of this normalizer sets the number of times it will serialize the same object before considering it a circular reference. Its default value is `1`.

Instead of throwing an exception, circular references can also be handled by custom callables. This is especially useful when serializing entities having unique identifiers:

Listing 83-19

```php
1  $encoder = new JsonEncoder();
2  $normalizer = new ObjectNormalizer();
3
4  $normalizer->setCircularReferenceHandler(function ($object) {
5      return $object->getName();
6  });
7
8  $serializer = new Serializer(array($normalizer), array($encoder));
9  var_dump($serializer->serialize($org, 'json'));
10 // {"name":"Les-Tilleuls.coop","members":[{"name":"K\u00e9vin", organization:
   "Les-Tilleuls.coop"}]}
```

Handling Arrays

The Serializer component is capable of handling arrays of objects as well. Serializing arrays works just like serializing a single object:

Listing 83-20

```php
1  use Acme\Person;
2
3  $person1 = new Person();
4  $person1->setName('foo');
5  $person1->setAge(99);
6  $person1->setSportsman(false);
7
8  $person2 = new Person();
9  $person2->setName('bar');
10 $person2->setAge(33);
11 $person2->setSportsman(true);
12
13 $persons = array($person1, $person2);
14 $data = $serializer->serialize($persons, 'json');
15
16 // $data contains
   [{"name":"foo","age":99,"sportsman":false},{"name":"bar","age":33,"sportsman":true}]
```

If you want to deserialize such a structure, you need to add the *ArrayDenormalizer*[19] to the set of normalizers. By appending `[]` to the type parameter of the *deserialize()*[20] method, you indicate that you're expecting an array instead of a single object.

Listing 83-21

```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ArrayDenormalizer;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;
use Symfony\Component\Serializer\Serializer;

$serializer = new Serializer(
    array(new GetSetMethodNormalizer(), new ArrayDenormalizer()),
    array(new JsonEncoder())
);

$data = ...; // The serialized data from the previous example
$persons = $serializer->deserialize($data, 'Acme\Person[]', 'json');
```

A popular alternative to the Symfony Serializer Component is the third-party library, JMS serializer[21] *(released under the Apache license, so incompatible with GPLv2 projects).*

19. http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/ArrayDenormalizer.html
20. http://api.symfony.com/3.0/Symfony/Component/Serializer/Serializer.html#method_deserialize
21. *https://github.com/schmittjoh/serializer*

Chapter 84
The Stopwatch Component

> The Stopwatch component provides a way to profile code.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/stopwatch` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/stopwatch*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The Stopwatch component provides an easy and consistent way to measure execution time of certain parts of code so that you don't constantly have to parse microtime by yourself. Instead, use the simple *`Stopwatch`*[2] class:

Listing 84-1

```
use Symfony\Component\Stopwatch\Stopwatch;

$stopwatch = new Stopwatch();
// Start event named 'eventName'
$stopwatch->start('eventName');
// ... some code goes here
$event = $stopwatch->stop('eventName');
```

1. https://packagist.org/packages/symfony/stopwatch
2. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html

The *StopwatchEvent*[3] object can be retrieved from the *start()*[4], *stop()*[5], *lap()*[6] and *getEvent()*[7] methods. The latter should be used when you need to retrieve the duration of an event while it is still running.

You can also provide a category name to an event:

Listing 84-2
```
$stopwatch->start('eventName', 'categoryName');
```

You can consider categories as a way of tagging events. For example, the Symfony Profiler tool uses categories to nicely color-code different events.

Periods

As you know from the real world, all stopwatches come with two buttons: one to start and stop the stopwatch, and another to measure the lap time. This is exactly what the *lap()*[8] method does:

Listing 84-3
```
1 $stopwatch = new Stopwatch();
2 // Start event named 'foo'
3 $stopwatch->start('foo');
4 // ... some code goes here
5 $stopwatch->lap('foo');
6 // ... some code goes here
7 $stopwatch->lap('foo');
8 // ... some other code goes here
9 $event = $stopwatch->stop('foo');
```

Lap information is stored as "periods" within the event. To get lap information call:

Listing 84-4
```
$event->getPeriods();
```

In addition to periods, you can get other useful information from the event object. For example:

Listing 84-5
```
1 $event->getCategory();   // Returns the category the event was started in
2 $event->getOrigin();     // Returns the event start time in milliseconds
3 $event->ensureStopped(); // Stops all periods not already stopped
4 $event->getStartTime();  // Returns the start time of the very first period
5 $event->getEndTime();    // Returns the end time of the very last period
6 $event->getDuration();   // Returns the event duration, including all periods
7 $event->getMemory();     // Returns the max memory usage of all periods
```

Sections

Sections are a way to logically split the timeline into groups. You can see how Symfony uses sections to nicely visualize the framework lifecycle in the Symfony Profiler tool. Here is a basic usage example using sections:

Listing 84-6

3. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/StopwatchEvent.html
4. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_start
5. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_stop
6. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_lap
7. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_getEvent
8. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_lap

```
1 $stopwatch = new Stopwatch();
2
3 $stopwatch->openSection();
4 $stopwatch->start('parsing_config_file', 'filesystem_operations');
5 $stopwatch->stopSection('routing');
6
7 $events = $stopwatch->getSectionEvents('routing');
```

You can reopen a closed section by calling the *openSection()*[9] method and specifying the id of the section to be reopened:

Listing 84-7

```
$stopwatch->openSection('routing');
$stopwatch->start('building_config_tree');
$stopwatch->stopSection('routing');
```

9. http://api.symfony.com/3.0/Symfony/Component/Stopwatch/Stopwatch.html#method_openSection

Chapter 85

The Templating Component

> The Templating component provides all the tools needed to build any kind of template system.
>
> It provides an infrastructure to load template files and optionally monitor them for changes. It also provides a concrete template engine implementation using PHP with additional tools for escaping and separating templates into blocks and layouts.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/templating` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/templating*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Usage

The *PhpEngine*[2] class is the entry point of the component. It needs a template name parser (*TemplateNameParserInterface*[3]) to convert a template name to a template reference (*TemplateReferenceInterface*[4]). It also needs a template loader (*LoaderInterface*[5]) which uses the template reference to actually find and load the template:

Listing 85-1

1. https://packagist.org/packages/symfony/templating
2. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html
3. http://api.symfony.com/3.0/Symfony/Component/Templating/TemplateNameParserInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Templating/TemplateReferenceInterface.html
5. http://api.symfony.com/3.0/Symfony/Component/Templating/Loader/LoaderInterface.html

```php
use Symfony\Component\Templating\PhpEngine;
use Symfony\Component\Templating\TemplateNameParser;
use Symfony\Component\Templating\Loader\FilesystemLoader;

$loader = new FilesystemLoader(__DIR__.'/views/%name%');

$templating = new PhpEngine(new TemplateNameParser(), $loader);

echo $templating->render('hello.php', array('firstname' => 'Fabien'));
```

Listing 85-2
```php
<!-- views/hello.php -->
Hello, <?php echo $firstname ?>!
```

The *render()*[6] method parses the `views/hello.php` file and returns the output text. The second argument of `render` is an array of variables to use in the template. In this example, the result will be `Hello, Fabien!`.

Templates will be cached in the memory of the engine. This means that if you render the same template multiple times in the same request, the template will only be loaded once from the file system.

The `$view` Variable

In all templates parsed by the `PhpEngine`, you get access to a mysterious variable called `$view`. That variable holds the current `PhpEngine` instance. That means you get access to a bunch of methods that make your life easier.

Including Templates

The best way to share a snippet of template code is to create a template that can then be included by other templates. As the `$view` variable is an instance of `PhpEngine`, you can use the `render` method (which was used to render the template originally) inside the template to render another template:

Listing 85-3
```php
<?php $names = array('Fabien', ...) ?>
<?php foreach ($names as $name) : ?>
    <?php echo $view->render('hello.php', array('firstname' => $name)) ?>
<?php endforeach ?>
```

Global Variables

Sometimes, you need to set a variable which is available in all templates rendered by an engine (like the `$app` variable when using the Symfony Framework). These variables can be set by using the *addGlobal()*[7] method and they can be accessed in the template as normal variables:

Listing 85-4
```php
$templating->addGlobal('ga_tracking', 'UA-xxxxx-x');
```

6. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_render
7. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_addGlobal

In a template:

Listing 85-5
```
1 <p>The google tracking code is: <?php echo $ga_tracking ?></p>
```

The global variables cannot be called `this` or `view`, since they are already used by the PHP engine.

The global variables can be overridden by a local variable in the template with the same name.

Output Escaping

When you render variables, you should probably escape them so that HTML or JavaScript code isn't written out to your page. This will prevent things like XSS attacks. To do this, use the *escape()*[8] method:

Listing 85-6
```
<?php echo $view->escape($firstname) ?>
```

By default, the `escape()` method assumes that the variable is outputted within an HTML context. The second argument lets you change the context. For example, to output something inside JavaScript, use the `js` context:

Listing 85-7
```
<?php echo $view->escape($var, 'js') ?>
```

The component comes with an HTML and JS escaper. You can register your own escaper using the *setEscaper()*[9] method:

Listing 85-8
```
1 $templating->setEscaper('css', function ($value) {
2     // ... all CSS escaping
3
4     return $escapedValue;
5 });
```

Helpers

The Templating component can be easily extended via helpers. Helpers are PHP objects that provide features useful in a template context. The component has 2 built-in helpers:

- *Assets Helper*
- *Slots Helper*

Before you can use these helpers, you need to register them using *set()*[10]:

Listing 85-9
```
use Symfony\Component\Templating\Helper\AssetsHelper;
// ...
```

8. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_escape
9. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_setEscaper
10. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_set

```
$templating->set(new AssetsHelper());
```

Custom Helpers

You can create your own helpers by creating a class which implements *HelperInterface*[11]. However, most of the time you'll extend *Helper*[12].

The `Helper` has one required method: *getName()*[13]. This is the name that is used to get the helper from the `$view` object.

Creating a Custom Engine

Besides providing a PHP templating engine, you can also create your own engine using the Templating component. To do that, create a new class which implements the *EngineInterface*[14]. This requires 3 method:

- *render($name, array $parameters = array())*[15] - Renders a template
- *exists($name)*[16] - Checks if the template exists
- *supports($name)*[17] - Checks if the given template can be handled by this engine.

Using Multiple Engines

It is possible to use multiple engines at the same time using the *DelegatingEngine*[18] class. This class takes a list of engines and acts just like a normal templating engine. The only difference is that it delegates the calls to one of the other engines. To choose which one to use for the template, the *EngineInterface::supports()*[19] method is used.

Listing 85-10

```
use Acme\Templating\CustomEngine;
use Symfony\Component\Templating\PhpEngine;
use Symfony\Component\Templating\DelegatingEngine;

$templating = new DelegatingEngine(array(
    new PhpEngine(...),
    new CustomEngine(...),
));
```

11. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/HelperInterface.html
12. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/Helper.html
13. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/HelperInterface.html#method_getName
14. http://api.symfony.com/3.0/Symfony/Component/Templating/EngineInterface.html
15. http://api.symfony.com/3.0/Symfony/Component/Templating/EngineInterface.html#method_render
16. http://api.symfony.com/3.0/Symfony/Component/Templating/EngineInterface.html#method_exists
17. http://api.symfony.com/3.0/Symfony/Component/Templating/EngineInterface.html#method_supports
18. http://api.symfony.com/3.0/Symfony/Component/Templating/DelegatingEngine.html
19. http://api.symfony.com/3.0/Symfony/Component/Templating/EngineInterface.html#method_supports

Chapter 86
Slots Helper

More often than not, templates in a project share common elements, like the well-known header and footer. Using this helper, the static HTML code can be placed in a layout file along with "slots", which represent the dynamic parts that will change on a page-by-page basis. These slots are then filled in by different children template. In other words, the layout file decorates the child template.

Displaying Slots

The slots are accessible by using the slots helper (`$view['slots']`). Use *`output()`*[1] to display the content of the slot on that place:

Listing 86-1

```
<!-- views/layout.php -->
<!doctype html>
<html>
    <head>
        <title>
            <?php $view['slots']->output('title', 'Default title') ?>
        </title>
    </head>
    <body>
        <?php $view['slots']->output('_content') ?>
    </body>
</html>
```

The first argument of the method is the name of the slot. The method has an optional second argument, which is the default value to use if the slot is not available.

The `_content` slot is a special slot set by the `PhpEngine`. It contains the content of the subtemplate.

1. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/SlotsHelper.html#method_output

If you're using the standalone component, make sure you registered the *SlotsHelper*[2]:

Listing 86-2
```
use Symfony\Component\Templating\Helper\SlotsHelper;

// ...
$templateEngine->set(new SlotsHelper());
```

Extending Templates

The *extend()*[3] method is called in the sub-template to set its parent template. Then *$view['slots']->set()*[4] can be used to set the content of a slot. All content which is not explicitly set in a slot is in the `_content` slot.

Listing 86-3
```
<!-- views/page.php -->
<?php $view->extend('layout.php') ?>

<?php $view['slots']->set('title', $page->title) ?>

<h1>
    <?php echo $page->title ?>
</h1>
<p>
    <?php echo $page->body ?>
</p>
```

Multiple levels of inheritance is possible: a layout can extend another layout.

For large slots, there is also an extended syntax:

Listing 86-4
```
<?php $view['slots']->start('title') ?>
    Some large amount of HTML
<?php $view['slots']->stop() ?>
```

2. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/SlotsHelper.html
3. http://api.symfony.com/3.0/Symfony/Component/Templating/PhpEngine.html#method_extend
4. http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/SlotsHelper.html#method_set

Chapter 87
Assets Helper

The assets helper's main purpose is to make your application more portable by generating asset paths:

Listing 87-1
```
1 <link href="<?php echo $view['assets']->getUrl('css/style.css') ?>" rel="stylesheet">
2
3 <img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>">
```

The assets helper can then be configured to render paths to a CDN or modify the paths in case your assets live in a sub-directory of your host (e.g. `http://example.com/app`).

Configure Paths

By default, the assets helper will prefix all paths with a slash. You can configure this by passing a base assets path as the first argument of the constructor:

Listing 87-2
```
use Symfony\Component\Templating\Helper\AssetsHelper;

// ...
$templateEngine->set(new AssetsHelper('/foo/bar'));
```

Now, if you use the helper, everything will be prefixed with `/foo/bar`:

Listing 87-3
```
1 <img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>">
2 <!-- renders as:
3 <img src="/foo/bar/images/logo.png">
4 -->
```

Absolute Urls

You can also specify a URL to use in the second parameter of the constructor:

Listing 87-4

```
// ...
$templateEngine->set(new AssetsHelper(null, 'http://cdn.example.com/'));
```

Now URLs are rendered like `http://cdn.example.com/images/logo.png`.

You can also use the third argument of the helper to force an absolute URL:

Listing 87-5

```
1 <img src="<?php echo $view['assets']->getUrl('images/logo.png', null, true) ?>">
2 <!-- renders as:
3 <img src="http://yourwebsite.com/foo/bar/images/logo.png">
4 -->
```

If you already set a URL in the constructor, using the third argument of `getUrl` will not affect the generated URL.

Versioning

To avoid using the cached resource after updating the old resource, you can use versions which you bump every time you release a new project. The version can be specified in the third argument:

Listing 87-6

```
// ...
$templateEngine->set(new AssetsHelper(null, null, '328rad75'));
```

Now, every URL is suffixed with `?328rad75`. If you want to have a different format, you can specify the new format in fourth argument. It's a string that is used in *sprintf*[1]. The first argument is the path and the second is the version. For instance, `%s?v=%s` will be rendered as `/images/logo.png?v=328rad75`.

You can also generate a versioned URL on an asset-by-asset basis using the fourth argument of the helper:

Listing 87-7

```
1 <img src="<?php echo $view['assets']->getUrl('images/logo.png', null, false, '3.0') ?>">
2 <!-- renders as:
3 <img src="/images/logo.png?v=3.0">
4 -->
```

Multiple Packages

Asset path generation is handled internally by packages. The component provides 2 packages by default:

- *PathPackage*[2]
- *UrlPackage*[3]

You can also use multiple packages:

Listing 87-8

```
1 use Symfony\Component\Templating\Asset\PathPackage;
2
3 // ...
4 $templateEngine->set(new AssetsHelper());
5
```

1. http://php.net/manual/en/function.sprintf.php
2. http://api.symfony.com/3.0/Symfony/Component/Templating/Asset/PathPackage.html
3. http://api.symfony.com/3.0/Symfony/Component/Templating/Asset/UrlPackage.html

```
6 $templateEngine->get('assets')->addPackage('images', new PathPackage('/images/'));
7 $templateEngine->get('assets')->addPackage('scripts', new PathPackage('/scripts/'));
```

This will setup the assets helper with 3 packages: the default package which defaults to `/` (set by the constructor), the images package which prefixes it with `/images/` and the scripts package which prefixes it with `/scripts/`.

If you want to set another default package, you can use *`setDefaultPackage()`*[4].

You can specify which package you want to use in the second argument of *`getUrl()`*[5]:

Listing 87-9

```
1 <img src="<?php echo $view['assets']->getUrl('foo.png', 'images') ?>">
2 <!-- renders as:
3 <img src="/images/foo.png">
4 -->
```

Custom Packages

You can create your own package by extending *`Package`*[6].

4. `http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/AssetsHelper.html#method_setDefaultPackage`
5. `http://api.symfony.com/3.0/Symfony/Component/Templating/Helper/AssetsHelper.html#method_getUrl`
6. `http://api.symfony.com/3.0/Symfony/Component/Templating/Asset/Package.html`

Chapter 88
The Translation Component

The Translation component provides tools to internationalize your application.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/translation` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/translation*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

Constructing the Translator

The main access point of the Translation component is *`Translator`*[2]. Before you can use it, you need to configure it and load the messages to translate (called *message catalogs*).

Configuration

The constructor of the `Translator` class needs one argument: The locale.

Listing 88-1
```
1  use Symfony\Component\Translation\Translator;
2  use Symfony\Component\Translation\MessageSelector;
3
4  $translator = new Translator('fr_FR', new MessageSelector());
```

1. `https://packagist.org/packages/symfony/translation`
2. `http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html`

The locale set here is the default locale to use. You can override this locale when translating strings.

The term *locale* refers roughly to the user's language and country. It can be any string that your application uses to manage translations and other format differences (e.g. currency format). The *ISO 639-1*[3] *language* code, an underscore (_), then the *ISO 3166-1 alpha-2*[4] *country* code (e.g. `fr_FR` for French/France) is recommended.

Loading Message Catalogs

The messages are stored in message catalogs inside the `Translator` class. A message catalog is like a dictionary of translations for a specific locale.

The Translation component uses Loader classes to load catalogs. You can load multiple resources for the same locale, which will then be combined into one catalog.

The component comes with some default Loaders and you can create your own Loader too. The default loaders are:

- *`ArrayLoader`*[5] - to load catalogs from PHP arrays.
- *`CsvFileLoader`*[6] - to load catalogs from CSV files.
- *`IcuDatFileLoader`*[7] - to load catalogs from resource bundles.
- *`IcuResFileLoader`*[8] - to load catalogs from resource bundles.
- *`IniFileLoader`*[9] - to load catalogs from ini files.
- *`MoFileLoader`*[10] - to load catalogs from gettext files.
- *`PhpFileLoader`*[11] - to load catalogs from PHP files.
- *`PoFileLoader`*[12] - to load catalogs from gettext files.
- *`QtFileLoader`*[13] - to load catalogs from QT XML files.
- *`XliffFileLoader`*[14] - to load catalogs from Xliff files.
- *`JsonFileLoader`*[15] - to load catalogs from JSON files.
- *`YamlFileLoader`*[16] - to load catalogs from Yaml files (requires the *Yaml component*).

All file loaders require the *Config component*.

You can also *create your own Loader*, in case the format is not already supported by one of the default loaders.

At first, you should add one or more loaders to the `Translator`:

Listing 88-2
```
// ...
$translator->addLoader('array', new ArrayLoader());
```

3. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
4. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
5. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/ArrayLoader.html
6. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/CsvFileLoader.html
7. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/IcuDatFileLoader.html
8. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/IcuResFileLoader.html
9. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/IniFileLoader.html
10. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/MoFileLoader.html
11. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/PhpFileLoader.html
12. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/PoFileLoader.html
13. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/QtFileLoader.html
14. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/XliffFileLoader.html
15. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/JsonFileLoader.html
16. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/YamlFileLoader.html

The first argument is the name to which you can refer the loader in the translator and the second argument is an instance of the loader itself. After this, you can add your resources using the correct loader.

Loading Messages with the `ArrayLoader`

Loading messages can be done by calling *`addResource()`*[17]. The first argument is the loader name (this was the first argument of the `addLoader` method), the second is the resource and the third argument is the locale:

Listing 88-3
```
// ...
$translator->addResource('array', array(
    'Hello World!' => 'Bonjour',
), 'fr_FR');
```

Loading Messages with the File Loaders

If you use one of the file loaders, you should also use the `addResource` method. The only difference is that you should put the file name to the resource file as the second argument, instead of an array:

Listing 88-4
```
// ...
$translator->addLoader('yaml', new YamlFileLoader());
$translator->addResource('yaml', 'path/to/messages.fr.yml', 'fr_FR');
```

The Translation Process

To actually translate the message, the Translator uses a simple process:

- A catalog of translated messages is loaded from translation resources defined for the `locale` (e.g. `fr_FR`). Messages from the Fallback Locales are also loaded and added to the catalog, if they don't already exist. The end result is a large "dictionary" of translations;
- If the message is located in the catalog, the translation is returned. If not, the translator returns the original message.

You start this process by calling *`trans()`*[18] or *`transChoice()`*[19]. Then, the Translator looks for the exact string inside the appropriate message catalog and returns it (if it exists).

Fallback Locales

If the message is not located in the catalog of the specific locale, the translator will look into the catalog of one or more fallback locales. For example, assume you're trying to translate into the `fr_FR` locale:

1. First, the translator looks for the translation in the `fr_FR` locale;
2. If it wasn't found, the translator looks for the translation in the `fr` locale;
3. If the translation still isn't found, the translator uses the one or more fallback locales set explicitly on the translator.

For (3), the fallback locales can be set by calling *`setFallbackLocales()`*[20]:

Listing 88-5
```
// ...
$translator->setFallbackLocales(array('en'));
```

17. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_addResource
18. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_trans
19. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_transChoice
20. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_setFallbackLocales

Using Message Domains

As you've seen, message files are organized into the different locales that they translate. The message files can also be organized further into "domains".

The domain is specified in the fourth argument of the `addResource()` method. The default domain is `messages`. For example, suppose that, for organization, translations were split into three different domains: `messages`, `admin` and `navigation`. The French translation would be loaded like this:

Listing 88-6

```
// ...
$translator->addLoader('xlf', new XliffFileLoader());

$translator->addResource('xlf', 'messages.fr.xlf', 'fr_FR');
$translator->addResource('xlf', 'admin.fr.xlf', 'fr_FR', 'admin');
$translator->addResource(
    'xlf',
    'navigation.fr.xlf',
    'fr_FR',
    'navigation'
);
```

When translating strings that are not in the default domain (`messages`), you must specify the domain as the third argument of `trans()`:

Listing 88-7

```
$translator->trans('Symfony is great', array(), 'admin');
```

Symfony will now look for the message in the `admin` domain of the specified locale.

Usage

Read how to use the Translation component in *Using the Translator*.

Chapter 89

Using the Translator

Imagine you want to translate the string *"Symfony is great"* into French:

Listing 89-1
```
use Symfony\Component\Translation\Translator;
use Symfony\Component\Translation\Loader\ArrayLoader;

$translator = new Translator('fr_FR');
$translator->addLoader('array', new ArrayLoader());
$translator->addResource('array', array(
    'Symfony is great!' => 'J\'aime Symfony!',
), 'fr_FR');

var_dump($translator->trans('Symfony is great!'));
```

In this example, the message *"Symfony is great!"* will be translated into the locale set in the constructor (`fr_FR`) if the message exists in one of the message catalogs.

Message Placeholders

Sometimes, a message containing a variable needs to be translated:

Listing 89-2
```
// ...
$translated = $translator->trans('Hello '.$name);

var_dump($translated);
```

However, creating a translation for this string is impossible since the translator will try to look up the exact message, including the variable portions (e.g. *"Hello Ryan"* or *"Hello Fabien"*). Instead of writing a translation for every possible iteration of the `$name` variable, you can replace the variable with a "placeholder":

Listing 89-3
```
// ...
$translated = $translator->trans(
    'Hello %name%',
```

```
4         array('%name%' => $name)
5     );
6
7     var_dump($translated);
```

Symfony will now look for a translation of the raw message (`Hello %name%`) and *then* replace the placeholders with their values. Creating a translation is done just as before:

Listing 89-4

```
1  <?xml version="1.0"?>
2  <xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
3      <file source-language="en" datatype="plaintext" original="file.ext">
4          <body>
5              <trans-unit id="1">
6                  <source>Hello %name%</source>
7                  <target>Bonjour %name%</target>
8              </trans-unit>
9          </body>
10     </file>
11 </xliff>
```

The placeholders can take on any form as the full message is reconstructed using the PHP *strtr function*[1]. But the `%...%` form is recommended, to avoid problems when using Twig.

As you've seen, creating a translation is a two-step process:

1. Abstract the message that needs to be translated by processing it through the `Translator`.
2. Create a translation for the message in each locale that you choose to support.

The second step is done by creating message catalogs that define the translations for any number of different locales.

Creating Translations

The act of creating translation files is an important part of "localization" (often abbreviated *L10n*[2]). Translation files consist of a series of id-translation pairs for the given domain and locale. The source is the identifier for the individual translation, and can be the message in the main locale (e.g. *"Symfony is great"*) of your application or a unique identifier (e.g. `symfony.great` - see the sidebar below).

Translation files can be created in several different formats, XLIFF being the recommended format. These files are parsed by one of the loader classes.

Listing 89-5

```
1  <?xml version="1.0"?>
2  <xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
3      <file source-language="en" datatype="plaintext" original="file.ext">
4          <body>
5              <trans-unit id="symfony_is_great">
6                  <source>Symfony is great</source>
7                  <target>J'aime Symfony</target>
8              </trans-unit>
9              <trans-unit id="symfony.great">
10                 <source>symfony.great</source>
```

1. http://php.net/manual/en/function.strtr.php
2. https://en.wikipedia.org/wiki/Internationalization_and_localization

```
11                 <target>J'aime Symfony</target>
12             </trans-unit>
13         </body>
14     </file>
15 </xliff>
```

Using Real or Keyword Messages

This example illustrates the two different philosophies when creating messages to be translated:

Listing 89-6
```
$translator->trans('Symfony is great');

$translator->trans('symfony.great');
```

In the first method, messages are written in the language of the default locale (English in this case). That message is then used as the "id" when creating translations.

In the second method, messages are actually "keywords" that convey the idea of the message. The keyword message is then used as the "id" for any translations. In this case, translations must be made for the default locale (i.e. to translate `symfony.great` to `Symfony is great`).

The second method is handy because the message key won't need to be changed in every translation file if you decide that the message should actually read "Symfony is really great" in the default locale.

The choice of which method to use is entirely up to you, but the "keyword" format is often recommended.

Additionally, the `php` and `yaml` file formats support nested ids to avoid repeating yourself if you use keywords instead of real text for your ids:

Listing 89-7
```
1 symfony:
2     is:
3         great: Symfony is great
4         amazing: Symfony is amazing
5     has:
6         bundles: Symfony has bundles
7 user:
8     login: Login
```

The multiple levels are flattened into single id/translation pairs by adding a dot (`.`) between every level, therefore the above examples are equivalent to the following:

Listing 89-8
```
1 symfony.is.great: Symfony is great
2 symfony.is.amazing: Symfony is amazing
3 symfony.has.bundles: Symfony has bundles
4 user.login: Login
```

Pluralization

Message pluralization is a tough topic as the rules can be quite complex. For instance, here is the mathematical representation of the Russian pluralization rules:

Listing 89-9
```
1 (($number % 10 == 1) && ($number % 100 != 11))
2     ? 0
```

```
3       : ((($number % 10 >= 2)
4           && ($number % 10 <= 4)
5           && (($number % 100 < 10)
6           || ($number % 100 >= 20)))
7               ? 1
8               : 2
9   );
```

As you can see, in Russian, you can have three different plural forms, each given an index of 0, 1 or 2. For each form, the plural is different, and so the translation is also different.

When a translation has different forms due to pluralization, you can provide all the forms as a string separated by a pipe (|):

Listing 89-10
```
'There is one apple|There are %count% apples'
```

To translate pluralized messages, use the *transChoice()*[3] method:

Listing 89-11
```
1 $translator->transChoice(
2     'There is one apple|There are %count% apples',
3     10,
4     array('%count%' => 10)
5 );
```

The second argument (`10` in this example) is the *number* of objects being described and is used to determine which translation to use and also to populate the `%count%` placeholder.

Based on the given number, the translator chooses the right plural form. In English, most words have a singular form when there is exactly one object and a plural form for all other numbers (0, 2, 3...). So, if `count` is `1`, the translator will use the first string (`There is one apple`) as the translation. Otherwise it will use `There are %count% apples`.

Here is the French translation:

Listing 89-12
```
1 'Il y a %count% pomme|Il y a %count% pommes'
```

Even if the string looks similar (it is made of two sub-strings separated by a pipe), the French rules are different: the first form (no plural) is used when `count` is `0` or `1`. So, the translator will automatically use the first string (`Il y a %count% pomme`) when `count` is `0` or `1`.

Each locale has its own set of rules, with some having as many as six different plural forms with complex rules behind which numbers map to which plural form. The rules are quite simple for English and French, but for Russian, you'd may want a hint to know which rule matches which string. To help translators, you can optionally "tag" each string:

Listing 89-13
```
1 'one: There is one apple|some: There are %count% apples'
2
3 'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

The tags are really only hints for translators and don't affect the logic used to determine which plural form to use. The tags can be any descriptive string that ends with a colon (`:`). The tags also do not need to be the same in the original message as in the translated one.

3. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_transChoice

As tags are optional, the translator doesn't use them (the translator will only get a string based on its position in the string).

Explicit Interval Pluralization

The easiest way to pluralize a message is to let the Translator use internal logic to choose which string to use based on a given number. Sometimes, you'll need more control or want a different translation for specific cases (for `0`, or when the count is negative, for example). For such cases, you can use explicit math intervals:

Listing 89-14
```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf[
There are many apples'
```

The intervals follow the *ISO 31-11*[4] notation. The above string specifies four different intervals: exactly `0`, exactly `1`, `2-19`, and `20` and higher.

You can also mix explicit math rules and standard rules. In this case, if the count is not matched by a specific interval, the standard rules take effect after removing the explicit rules:

Listing 89-15
```
'{0} There are no apples|[20,Inf[ There are many apples|There is one apple|a_few: There are
%count% apples'
```

For example, for `1` apple, the standard rule `There is one apple` will be used. For `2-19` apples, the second standard rule `There are %count% apples` will be selected.

An *`Interval`*[5] can represent a finite set of numbers:

Listing 89-16
```
{1,2,3,4}
```

Or numbers between two other numbers:

Listing 89-17
```
[1, +Inf[
]-1,2[
```

The left delimiter can be `[` (inclusive) or `]` (exclusive). The right delimiter can be `[` (exclusive) or `]` (inclusive). Beside numbers, you can use `-Inf` and `+Inf` for the infinite.

Forcing the Translator Locale

When translating a message, the Translator uses the specified locale or the `fallback` locale if necessary. You can also manually specify the locale to use for translation:

Listing 89-18
```
$translator->trans(
    'Symfony is great',
    array(),
    'messages',
    'fr_FR'
);

```

4. https://en.wikipedia.org/wiki/Interval_(mathematics)#Notations_for_intervals
5. http://api.symfony.com/3.0/Symfony/Component/Translation/Interval.html

```
8  $translator->transChoice(
9      '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
10     10,
11     array('%count%' => 10),
12     'messages',
13     'fr_FR'
14 );
```

Retrieving the Message Catalogue

In case you want to use the same translation catalogue outside your application (e.g. use translation on the client side), it's possible to fetch raw translation messages. Just specify the required locale:

Listing 89-19
```
1 $catalogue = $translator->getCatalogue('fr_FR');
2 $messages = $catalogue->all();
3 while ($catalogue = $catalogue->getFallbackCatalogue()) {
4     $messages = array_replace_recursive($catalogue->all(), $messages);
5 }
```

The `$messages` variable will have the following structure:

Listing 89-20
```
1 array(
2     'messages' => array(
3         'Hello world' => 'Bonjour tout le monde',
4     ),
5     'validators' => array(
6         'Value should not be empty' => 'Valeur ne doit pas être vide',
7         'Value is too long' => 'Valeur est trop long',
8     ),
9 );
```

Chapter 90

Adding Custom Format Support

Sometimes, you need to deal with custom formats for translation files. The Translation component is flexible enough to support this. Just create a loader (to load translations) and, optionally, a dumper (to dump translations).

Imagine that you have a custom format where translation messages are defined using one line for each translation and parentheses to wrap the key and the message. A translation file would look like this:

Listing 90-1

```
(welcome)(accueil)
(goodbye)(au revoir)
(hello)(bonjour)
```

Creating a Custom Loader

To define a custom loader that is able to read these kinds of files, you must create a new class that implements the *LoaderInterface*[1]. The *load()*[2] method will get a filename and parse it into an array. Then, it will create the catalog that will be returned:

Listing 90-2

```
use Symfony\Component\Translation\MessageCatalogue;
use Symfony\Component\Translation\Loader\LoaderInterface;

class MyFormatLoader implements LoaderInterface
{
    public function load($resource, $locale, $domain = 'messages')
    {
        $messages = array();
        $lines = file($resource);

        foreach ($lines as $line) {
            if (preg_match('/\(([^\)]+)\)\(([^\)]+)\)/', $line, $matches)) {
                $messages[$matches[1]] = $matches[2];
```

1. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/LoaderInterface.html
2. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/LoaderInterface.html#method_load

```
14                  }
15              }
16
17              $catalogue = new MessageCatalogue($locale);
18              $catalogue->add($messages, $domain);
19
20              return $catalogue;
21          }
22
23      }
```

Once created, it can be used as any other loader:

Listing 90-3

```
1  use Symfony\Component\Translation\Translator;
2
3  $translator = new Translator('fr_FR');
4  $translator->addLoader('my_format', new MyFormatLoader());
5
6  $translator->addResource('my_format', __DIR__.'/translations/messages.txt', 'fr_FR');
7
8  var_dump($translator->trans('welcome'));
```

It will print *"accueil"*.

Creating a Custom Dumper

It is also possible to create a custom dumper for your format, which is useful when using the extraction commands. To do so, a new class implementing the *DumperInterface*[3] must be created. To write the dump contents into a file, extending the *FileDumper*[4] class will save a few lines:

Listing 90-4

```
1   use Symfony\Component\Translation\MessageCatalogue;
2   use Symfony\Component\Translation\Dumper\FileDumper;
3
4   class MyFormatDumper extends FileDumper
5   {
6       public function formatCatalogue(MessageCatalogue $messages, $domain, array $options =
7   array())
8       {
9           $output = '';
10
11          foreach ($messages->all($domain) as $source => $target) {
12              $output .= sprintf("(%s)(%s)\n", $source, $target);
13          }
14
15          return $output;
16      }
17
18      protected function getExtension()
19      {
20          return 'txt';
21      }
    }
```

3. http://api.symfony.com/3.0/Symfony/Component/Translation/Dumper/DumperInterface.html
4. http://api.symfony.com/3.0/Symfony/Component/Translation/Dumper/FileDumper.html

Format a message catalogue

In some cases, you want to send the dump contents as a response instead of writing them in files. To do this, you can use the `formatCatalogue` method. In this case, you must pass the domain argument, which determines the list of messages that should be dumped.

The *formatCatalogue()*[5] method creates the output string, that will be used by the *dump()*[6] method of the FileDumper class to create the file. The dumper can be used like any other built-in dumper. In the following example, the translation messages defined in the YAML file are dumped into a text file with the custom format:

Listing 90-5
```
1 use Symfony\Component\Translation\Loader\YamlFileLoader;
2
3 $loader = new YamlFileLoader();
4 $catalogue = $loader->load(__DIR__ . '/translations/messages.fr_FR.yml' , 'fr_FR');
5
6 $dumper = new MyFormatDumper();
7 $dumper->dump($catalogue, array('path' => __DIR__.'/dumps'));
```

5. http://api.symfony.com/3.0/Symfony/Component/Translation/Dumper/FileDumper.html#method_formatCatalogue
6. http://api.symfony.com/3.0/Symfony/Component/Translation/Dumper/FileDumper.html#method_dump

Chapter 91

The VarDumper Component

The VarDumper component provides mechanisms for walking through any arbitrary PHP variable. Built on top, it provides a better `dump()` function that you can use instead of *`var_dump`*[1].

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/var-dumper` on *Packagist*[2]);
- Use the official Git repository (*https://github.com/symfony/var-dumper*).

If using it inside a Symfony application, make sure that the DebugBundle is enabled in your `app/AppKernel.php` file.

The dump() Function

The VarDumper component creates a global `dump()` function that you can use instead of e.g. *`var_dump`*[3]. By using it, you'll gain:

- Per object and resource types specialized view to e.g. filter out Doctrine internals while dumping a single proxy entity, or get more insight on opened files with *`stream_get_meta_data`*[4];
- Configurable output formats: HTML or colored command line output;

1. http://php.net/manual/en/function.var-dump.php
2. https://packagist.org/packages/symfony/var-dumper
3. http://php.net/manual/en/function.var-dump.php
4. http://php.net/manual/en/function.stream-get-meta-data.php

- Ability to dump internal references, either soft ones (objects or resources) or hard ones (`=&` on arrays or objects properties). Repeated occurrences of the same object/array/resource won't appear again and again anymore. Moreover, you'll be able to inspect the reference structure of your data;
- Ability to operate in the context of an output buffering handler.

For example:

Listing 91-1

```php
require __DIR__.'/vendor/autoload.php';

// create a variable, which could be anything!
$someVar = ...;

dump($someVar);
```

By default, the output format and destination are selected based on your current PHP SAPI:

- On the command line (CLI SAPI), the output is written on `STDOUT`. This can be surprising to some because this bypasses PHP's output buffering mechanism;
- On other SAPIs, dumps are written as HTML in the regular output.

If you want to catch the dump output as a string, please read the *advanced documentation*[5] which contains examples of it. You'll also learn how to change the format or redirect the output to wherever you want.

In order to have the `dump()` function always available when running any PHP code, you can install it globally on your computer:

1. Run `composer global require symfony/var-dumper`;
2. Add `auto_prepend_file = ${HOME}/.composer/vendor/autoload.php` to your `php.ini` file;
3. From time to time, run `composer global update symfony/var-dumper` to have the latest bug fixes.

DebugBundle and Twig Integration

The DebugBundle allows greater integration of the component into the Symfony full-stack framework. It is enabled by default in the *dev* and *test* environment of the Symfony Standard Edition.

Since generating (even debug) output in the controller or in the model of your application may just break it by e.g. sending HTTP headers or corrupting your view, the bundle configures the `dump()` function so that variables are dumped in the web debug toolbar.

But if the toolbar can not be displayed because you e.g. called `die`/`exit` or a fatal error occurred, then dumps are written on the regular output.

In a Twig template, two constructs are available for dumping a variable. Choosing between both is mostly a matter of personal taste, still:

- `{% dump foo.bar %}` is the way to go when the original template output shall not be modified: variables are not dumped inline, but in the web debug toolbar;
- on the contrary, `{{ dump(foo.bar) }}` dumps inline and thus may or not be suited to your use case (e.g. you shouldn't use it in an HTML attribute or a `<script>` tag).

5. #components-var_dumper-advanced

This behavior can be changed by configuring the `dump.dump_destination` option. Read more about this and other options in *the DebugBundle configuration reference*.

Using the VarDumper Component in your PHPUnit Test Suite

The VarDumper component provides *a trait*[6] that can help writing some of your tests for PHPUnit.

This will provide you with two new assertions:

assertDumpEquals()[7]
: verifies that the dump of the variable given as the second argument matches the expected dump provided as a string in the first argument.

assertDumpMatchesFormat()[8]
: is like the previous method but accepts placeholders in the expected dump, based on the `assertStringMatchesFormat` method provided by PHPUnit.

Example:

Listing 91-2

```
class ExampleTest extends \PHPUnit_Framework_TestCase
{
    use \Symfony\Component\VarDumper\Test\VarDumperTestTrait;

    public function testWithDumpEquals()
    {
        $testedVar = array(123, 'foo');

        $expectedDump = <<<EOTXT
array:2 [
  0 => 123
  1 => "foo"
]
EOTXT;

        $this->assertDumpEquals($expectedDump, $testedVar);
    }
}
```

Dump Examples and Output

For simple variables, reading the output should be straightforward. Here are some examples showing first a variable defined in PHP, then its dump representation:

Listing 91-3

```
$var = array(
    'a simple string' => "in an array of 5 elements",
    'a float' => 1.0,
    'an integer' => 1,
    'a boolean' => true,
    'an empty array' => array(),
);
dump($var);
```

6. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Test/VarDumperTestTrait.html
7. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Test/VarDumperTestTrait.html#method_assertDumpEquals
8. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Test/VarDumperTestTrait.html#method_assertDumpMatchesFormat

```
array:5 [▼
  "a simple string" => "in an array of 5 elements"
  "a float" => 1.0
  "an integer" => 1
  "a boolean" => true
  "an empty array" => []
]
```

The gray arrow is a toggle button for hiding/showing children of nested structures.

Listing 91-4
```
1 $var = "This is a multi-line string.\n";
2 $var .= "Hovering a string shows its length.\n";
3 $var .= "The length of UTF-8 strings is counted in terms of UTF-8 characters.\n";
4 $var .= "Non-UTF-8 strings length are counted in octet size.\n";
5 $var .= "Because of this `\xE9` octet (\\xE9),\n";
6 $var .= "this string is not UTF-8 valid, thus the `b` prefix.\n";
7 dump($var);
```

```
b"""
  This is a multi-line string.
  Hovering a string shows its length.
  The length of UTF-8 strings is counted in terms of UTF-8 characters.
  Non-UTF-8 strings length are counted in octet size.
  Because of this `é` octet (\xE9),
  this string is not UTF-8 valid, thus the `b` prefix.
  """
```

Listing 91-5
```
1 class PropertyExample
2 {
3     public $publicProperty = 'The `+` prefix denotes public properties,';
4     protected $protectedProperty = '`#` protected ones and `-` private ones.';
5     private $privateProperty = 'Hovering a property shows a reminder.';
6 }
7
8 $var = new PropertyExample();
9 dump($var);
```

```
PropertyExample {#14 ▼
  +publicProperty: "The `+` prefix denotes public properties,"
  #protectedProperty: "`#` protected ones and `-` private ones."
  -privateProperty: "Hovering a property shows a reminder."
}
```

#14 is the internal object handle. It allows comparing two consecutive dumps of the same object.

Listing 91-6
```
1 class DynamicPropertyExample
2 {
3     public $declaredProperty = 'This property is declared in the class definition';
4 }
```

```
5
6 $var = new DynamicPropertyExample();
7 $var->undeclaredProperty = 'Runtime added dynamic properties have `"` around their name.';
8 dump($var);
```

```
DynamicPropertyExample {#15 ▼
  +declaredProperty: "This property is declared in the class definition"
  +"undeclaredProperty": "Runtime added dynamic properties have `"` around their name."
}
```

Listing 91-7

```
1 class ReferenceExample
2 {
3     public $info = "Circular and sibling references are displayed as `#number`.\nHovering
4 them highlights all instances in the same dump.\n";
5 }
6 $var = new ReferenceExample();
7 $var->aCircularReference = $var;
  dump($var);
```

```
ReferenceExample {#16 ▼
  +info: """
    Circular and sibling references are displayed as `#number`.
    Hovering them highlights all instances in the same dump.
    """
  +"aCircularReference": ReferenceExample {#16}
}
```

Listing 91-8

```
1 $var = new \ErrorException(
2     "For some objects, properties have special values\n"
3     ."that are best represented as constants, like\n"
4     ."`severity` below. Hovering displays the value (`2`).\n",
5     0,
6     E_WARNING
7 );
8 dump($var);
```

```
ErrorException {#14 ▼
  #message: """
    For some objects, properties have special values
    that are best represented as constants, like
    `severity` below. Hovering displays the value (`2`).
    """
  #code: 0
  #file: "…/var-dumper-example.php"
  #line: 58
  #severity: E_WARNING
  -trace: array:1 [▶]
}
```

Listing 91-9

```
1 $var = array();
2 $var[0] = 1;
3 $var[1] =& $var[0];
4 $var[1] += 1;
5 $var[2] = array("Hard references (circular or sibling)");
6 $var[3] =& $var[2];
7 $var[3][] = "are dumped using `&number` prefixes.";
8 dump($var);
```

```
array:4 [▼
  0 => &1 2
  1 => &1 2
  2 => &2 array:2 [▼
    0 => "Hard references (circular or sibling)"
    1 => "are dumped using `&number` prefixes."
  ]
  3 => &2 array:2 [▶]
]
```

Listing 91-10
```
$var = new \ArrayObject();
$var[] = "Some resources and special objects like the current";
$var[] = "one are sometimes best represented using virtual";
$var[] = "properties that describe their internal state.";
dump($var);
```

```
ArrayObject {#17 ▼
  flag::STD_PROP_LIST: false
  flag::ARRAY_AS_PROPS: false
  iteratorClass: "ArrayIterator"
  storage: array:3 [▼
    0 => "Some resources and special objects like the current"
    1 => "one are sometimes best represented using virtual"
    2 => "properties that describe their internal state."
  ]
}
```

Listing 91-11
```
$var = new AcmeController(
    "When a dump goes over its maximum items limit,\n"
    ."or when some special objects are encountered,\n"
    ."children can be replaced by an ellipsis and\n"
    ."optionally followed by a number that says how\n"
    ."many have been removed; `9` in this case.\n"
);
dump($var);
```

```
AcmeController {#14 ▼
  -info: """
    When a dump goes over its maximum items limit,
    or when some special objects are encountered,
    children can be replaced by an ellipsis and
    optionnally followed by a number that says how
    many have been removed; `9` in this case.
    """
  #container: Symfony\Component\DependencyInjection\Container {#15 …9}
}
```

Chapter 92

Advanced Usage of the VarDumper Component

The `dump()` function is just a thin wrapper and a more convenient way to call *`VarDumper::dump()`*[1]. You can change the behavior of this function by calling *`VarDumper::setHandler($callable)`*[2]. Calls to `dump()` will then be forwarded to `$callable`.

By adding a handler, you can customize the Cloners, Dumpers and Casters as explained below. A simple implementation of a handler function might look like this:

Listing 92-1

```
use Symfony\Component\VarDumper\VarDumper;
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;
use Symfony\Component\VarDumper\Dumper\HtmlDumper;

VarDumper::setHandler(function ($var) {
    $cloner = new VarCloner();
    $dumper = 'cli' === PHP_SAPI ? new CliDumper() : new HtmlDumper();

    $dumper->dump($cloner->cloneVar($var));
});
```

Cloners

A cloner is used to create an intermediate representation of any PHP variable. Its output is a *`Data`*[3] object that wraps this representation.

You can create a *`Data`*[4] object this way:

Listing 92-2

```
use Symfony\Component\VarDumper\Cloner\VarCloner;

$cloner = new VarCloner();
```

1. http://api.symfony.com/3.0/Symfony/Component/VarDumper/VarDumper.html#method_dump
2. http://api.symfony.com/3.0/Symfony/Component/VarDumper/VarDumper.html#method_setHandler
3. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
4. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html

```
4 $data = $cloner->cloneVar($myVar);
5 // this is commonly then passed to the dumper
6 // see the example at the top of this page
7 // $dumper->dump($data);
```

A cloner also applies limits when creating the representation, so that the corresponding Data object could represent only a subset of the cloned variable. Before calling *cloneVar()*[5], you can configure these limits:

setMaxItems()[6]
: configures the maximum number of items that will be cloned *past the first nesting level*. Items are counted using a breadth-first algorithm so that lower level items have higher priority than deeply nested items;

setMaxString()[7]
: configures the maximum number of characters that will be cloned before cutting overlong strings;

In both cases, specifying `-1` removes any limit.

Before dumping it, you can further limit the resulting *Data*[8] object using the following methods:

withMaxDepth()[9]
: Allows limiting dumps in the depth dimension.

withMaxItemsPerDepth()[10]
: Limits the number of items per depth level.

withRefHandles()[11]
: Allows removing internal objects' handles for sparser output (useful for tests).

Unlike the previous limits on cloners that remove data on purpose, these can be changed back and forth before dumping since they do not affect the intermediate representation internally.

When no limit is applied, a *Data*[12] object is as accurate as the native *serialize*[13] function, and thus could be for purposes beyond dumping for debugging.

Dumpers

A dumper is responsible for outputting a string representation of a PHP variable, using a *Data*[14] object as input. The destination and the formatting of this output vary with dumpers.

This component comes with an *HtmlDumper*[15] for HTML output and a *CliDumper*[16] for optionally colored command line output.

For example, if you want to dump some `$variable`, just do:

5. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/VarCloner.html#method_cloneVar
6. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/VarCloner.html#method_setMaxItems
7. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/VarCloner.html#method_setMaxString
8. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
9. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html#method_withMaxDepth
10. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html#method_withMaxItemsPerDepth
11. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html#method_withRefHandles
12. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
13. http://php.net/manual/en/function.serialize.php
14. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
15. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/HtmlDumper.html
16. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/CliDumper.html

Listing 92-3

```php
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$cloner = new VarCloner();
$dumper = new CliDumper();

$dumper->dump($cloner->cloneVar($variable));
```

By using the first argument of the constructor, you can select the output stream where the dump will be written. By default, the `CliDumper` writes on `php://stdout` and the `HtmlDumper` on `php://output`. But any PHP stream (resource or URL) is acceptable.

Instead of a stream destination, you can also pass it a `callable` that will be called repeatedly for each line generated by a dumper. This callable can be configured using the first argument of a dumper's constructor, but also using the *`setOutput()`*[17] method or the second argument of the *`dump()`*[18] method.

For example, to get a dump as a string in a variable, you can do:

Listing 92-4

```php
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$cloner = new VarCloner();
$dumper = new CliDumper();
$output = '';

$dumper->dump(
    $cloner->cloneVar($variable),
    function ($line, $depth) use (&$output) {
        // A negative depth means "end of dump"
        if ($depth >= 0) {
            // Adds a two spaces indentation to the line
            $output .= str_repeat('  ', $depth).$line."\n";
        }
    }
);

// $output is now populated with the dump representation of $variable
```

Another option for doing the same could be:

Listing 92-5

```php
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$cloner = new VarCloner();
$dumper = new CliDumper();
$output = fopen('php://memory', 'r+b');

$dumper->dump($cloner->cloneVar($variable), $output);
$output = stream_get_contents($output, -1, 0);

// $output is now populated with the dump representation of $variable
```

17. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/AbstractDumper.html#method_setOutput
18. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/AbstractDumper.html#method_dump

Dumpers implement the *DataDumperInterface*[19] interface that specifies the *dump(Data $data)*[20] method. They also typically implement the *DumperInterface*[21] that frees them from re-implementing the logic required to walk through a *Data*[22] object's internal structure.

Casters

Objects and resources nested in a PHP variable are "cast" to arrays in the intermediate *Data*[23] representation. You can tweak the array representation for each object/resource by hooking a Caster into this process. The component already includes many casters for base PHP classes and other common classes.

If you want to build your own Caster, you can register one before cloning a PHP variable. Casters are registered using either a Cloner's constructor or its `addCasters()` method:

Listing 92-6
```
use Symfony\Component\VarDumper\Cloner\VarCloner;

$myCasters = array(...);
$cloner = new VarCloner($myCasters);

// or

$cloner->addCasters($myCasters);
```

The provided `$myCasters` argument is an array that maps a class, an interface or a resource type to a callable:

Listing 92-7
```
$myCasters = array(
    'FooClass' => $myFooClassCallableCaster,
    ':bar resource' => $myBarResourceCallableCaster,
);
```

As you can notice, resource types are prefixed by a `:` to prevent colliding with a class name.

Because an object has one main class and potentially many parent classes or interfaces, many casters can be applied to one object. In this case, casters are called one after the other, starting from casters bound to the interfaces, the parents classes and then the main class. Several casters can also be registered for the same resource type/class/interface. They are called in registration order.

Casters are responsible for returning the properties of the object or resource being cloned in an array. They are callables that accept four arguments:

- the object or resource being casted,
- an array modelled for objects after PHP's native `(array)` cast operator,
- a *Stub*[24] object representing the main properties of the object (class, type, etc.),
- true/false when the caster is called nested in a structure or not.

Here is a simple caster not doing anything:

Listing 92-8
```
function myCaster($object, $array, $stub, $isNested)
{
```

19. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/DataDumperInterface.html
20. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Dumper/DataDumperInterface.html#method_dump
21. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/DumperInterface.html
22. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
23. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Data.html
24. http://api.symfony.com/3.0/Symfony/Component/VarDumper/Cloner/Stub.html

```
3     // ... populate/alter $array to your needs
4
5     return $array;
6 }
```

For objects, the `$array` parameter comes pre-populated using PHP's native `(array)` casting operator or with the return value of `$object->__debugInfo()` if the magic method exists. Then, the return value of one Caster is given as the array argument to the next Caster in the chain.

When casting with the `(array)` operator, PHP prefixes protected properties with a `\0*\0` and private ones with the class owning the property. For example, `\0Foobar\0` will be the prefix for all private properties of objects of type Foobar. Casters follow this convention and add two more prefixes: `\0~\0` is used for virtual properties and `\0+\0` for dynamic ones (runtime added properties not in the class declaration).

Although you can, it is advised to not alter the state of an object while casting it in a Caster.

Before writing your own casters, you should check the existing ones.

Chapter 93
The Yaml Component

> The Yaml component loads and dumps YAML files.

What is It?

The Symfony Yaml component parses YAML strings to convert them to PHP arrays. It is also able to convert PHP arrays to YAML strings.

YAML[1], *YAML Ain't Markup Language*, is a human friendly data serialization standard for all programming languages. YAML is a great format for your configuration files. YAML files are as expressive as XML files and as readable as INI files.

The Symfony Yaml Component implements a selected subset of features defined in the *YAML 1.2 version specification*[2].

Learn more about the Yaml component in the *The YAML Format* article.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/yaml` on *Packagist*[3]);
- Use the official Git repository (*https://github.com/symfony/yaml*).

Then, require the `vendor/autoload.php` file to enable the autoloading mechanism provided by Composer. Otherwise, your application won't be able to find the classes of this Symfony component.

1. `http://yaml.org/`
2. `http://yaml.org/spec/1.2/spec.html`
3. `https://packagist.org/packages/symfony/yaml`

Why?

Fast

One of the goals of Symfony Yaml is to find the right balance between speed and features. It supports just the needed features to handle configuration files. Notable lacking features are: document directives, multi-line quoted messages, compact block collections and multi-document files.

Real Parser

It sports a real parser and is able to parse a large subset of the YAML specification, for all your configuration needs. It also means that the parser is pretty robust, easy to understand, and simple enough to extend.

Clear Error Messages

Whenever you have a syntax problem with your YAML files, the library outputs a helpful message with the filename and the line number where the problem occurred. It eases the debugging a lot.

Dump Support

It is also able to dump PHP arrays to YAML with object support, and inline level configuration for pretty outputs.

Types Support

It supports most of the YAML built-in types like dates, integers, octals, booleans, and much more...

Full Merge Key Support

Full support for references, aliases, and full merge key. Don't repeat yourself by referencing common configuration bits.

Using the Symfony YAML Component

The Symfony Yaml component is very simple and consists of two main classes: one parses YAML strings (*Parser*[4]), and the other dumps a PHP array to a YAML string (*Dumper*[5]).

On top of these two classes, the *Yaml*[6] class acts as a thin wrapper that simplifies common uses.

Reading YAML Files

The *parse()*[7] method parses a YAML string and converts it to a PHP array:

Listing 93-1

```
use Symfony\Component\Yaml\Parser;

$yaml = new Parser();
```

4. http://api.symfony.com/3.0/Symfony/Component/Yaml/Parser.html
5. http://api.symfony.com/3.0/Symfony/Component/Yaml/Dumper.html
6. http://api.symfony.com/3.0/Symfony/Component/Yaml/Yaml.html
7. http://api.symfony.com/3.0/Symfony/Component/Yaml/Parser.html#method_parse

```
4
5 $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
```

If an error occurs during parsing, the parser throws a *ParseException*[8] exception indicating the error type and the line in the original YAML string where the error occurred:

Listing 93-2
```
1 use Symfony\Component\Yaml\Exception\ParseException;
2
3 try {
4     $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
5 } catch (ParseException $e) {
6     printf("Unable to parse the YAML string: %s", $e->getMessage());
7 }
```

As the parser is re-entrant, you can use the same parser object to load different YAML strings.

It may also be convenient to use the *parse()*[9] wrapper method:

Listing 93-3
```
1 use Symfony\Component\Yaml\Yaml;
2
3 $yaml = Yaml::parse(file_get_contents('/path/to/file.yml'));
```

The *parse()*[10] static method takes a YAML string. Internally, it constructs a *Parser*[11] object and calls the *parse()*[12] method.

Writing YAML Files

The *dump()*[13] method dumps any PHP array to its YAML representation:

Listing 93-4
```
1  use Symfony\Component\Yaml\Dumper;
2
3  $array = array(
4      'foo' => 'bar',
5      'bar' => array('foo' => 'bar', 'bar' => 'baz'),
6  );
7
8  $dumper = new Dumper();
9
10 $yaml = $dumper->dump($array);
11
12 file_put_contents('/path/to/file.yml', $yaml);
```

8. http://api.symfony.com/3.0/Symfony/Component/Yaml/Exception/ParseException.html
9. http://api.symfony.com/3.0/Symfony/Component/Yaml/Yaml.html#method_parse
10. http://api.symfony.com/3.0/Symfony/Component/Yaml/Yaml.html#method_parse
11. http://api.symfony.com/3.0/Symfony/Component/Yaml/Parser.html
12. http://api.symfony.com/3.0/Symfony/Component/Yaml/Parser.html#method_parse
13. http://api.symfony.com/3.0/Symfony/Component/Yaml/Dumper.html#method_dump

Of course, the Symfony Yaml dumper is not able to dump resources. Also, even if the dumper is able to dump PHP objects, it is considered to be a not supported feature.

If an error occurs during the dump, the parser throws a *DumpException*[14] exception.

If you only need to dump one array, you can use the *dump()*[15] static method shortcut:

Listing 93-5
```
use Symfony\Component\Yaml\Yaml;

$yaml = Yaml::dump($array, $inline);
```

The YAML format supports two kind of representation for arrays, the expanded one, and the inline one. By default, the dumper uses the inline representation:

Listing 93-6
```
{ foo: bar, bar: { foo: bar, bar: baz } }
```

The second argument of the *dump()*[16] method customizes the level at which the output switches from the expanded representation to the inline one:

Listing 93-7
```
echo $dumper->dump($array, 1);
```

Listing 93-8
```
foo: bar
bar: { foo: bar, bar: baz }
```

Listing 93-9
```
echo $dumper->dump($array, 2);
```

Listing 93-10
```
foo: bar
bar:
    foo: bar
    bar: baz
```

14. http://api.symfony.com/3.0/Symfony/Component/Yaml/Exception/DumpException.html
15. http://api.symfony.com/3.0/Symfony/Component/Yaml/Yaml.html#method_dump
16. http://api.symfony.com/3.0/Symfony/Component/Yaml/Dumper.html#method_dump

Chapter 94

The YAML Format

According to the official *YAML*[1] website, YAML is "a human friendly data serialization standard for all programming languages".

Even if the YAML format can describe complex nested data structure, this chapter only describes the minimum set of features needed to use YAML as a configuration file format.

YAML is a simple language that describes data. As PHP, it has a syntax for simple types like strings, booleans, floats, or integers. But unlike PHP, it makes a difference between arrays (sequences) and hashes (mappings).

Scalars

The syntax for scalars is similar to the PHP syntax.

Strings

Strings in YAML can be wrapped both in single and double quotes. In some cases, they can also be unquoted:

Listing 94-1

```
A string in YAML

'A singled-quoted string in YAML'

"A double-quoted string in YAML"
```

Quoted styles are useful when a string starts or end with one or more relevant spaces, because unquoted strings are trimmed on both end when parsing their contents. Quotes are required when the string contains special or reserved characters.

When using single-quoted strings, any single quote `'` inside its contents must be doubled to escape it:

Listing 94-2

```
'A single quote '' inside a single-quoted string'
```

1. `http://yaml.org/`

Strings containing any of the following characters must be quoted. Although you can use double quotes, for these characters it is more convenient to use single quotes, which avoids having to escape any backslash `\`:

- `:`, `{`, `}`, `[`, `]`, `,`, `&`, `*`, `#`, `?`, `|`, `-`, `<`, `>`, `=`, `!`, `%`, `@`, `` ` ``

The double-quoted style provides a way to express arbitrary strings, by using `\` to escape characters and sequences. For instance, it is very useful when you need to embed a `\n` or a Unicode character in a string.

Listing 94-3
```
1 "A double-quoted string in YAML\n"
```

If the string contains any of the following control characters, it must be escaped with double quotes:

- `\0`, `\x01`, `\x02`, `\x03`, `\x04`, `\x05`, `\x06`, `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\x0e`, `\x0f`, `\x10`, `\x11`, `\x12`, `\x13`, `\x14`, `\x15`, `\x16`, `\x17`, `\x18`, `\x19`, `\x1a`, `\e`, `\x1c`, `\x1d`, `\x1e`, `\x1f`, `\N`, `_`, `\L`, `\P`

Finally, there are other cases when the strings must be quoted, no matter if you're using single or double quotes:

- When the string is `true` or `false` (otherwise, it would be treated as a boolean value);
- When the string is `null` or `~` (otherwise, it would be considered as a `null` value);
- When the string looks like a number, such as integers (e.g. `2`, `14`, etc.), floats (e.g. `2.6`, `14.9`) and exponential numbers (e.g. `12e7`, etc.) (otherwise, it would be treated as a numeric value);
- When the string looks like a date (e.g. `2014-12-31`) (otherwise it would be automatically converted into a Unix timestamp).

When a string contains line breaks, you can use the literal style, indicated by the pipe (`|`), to indicate that the string will span several lines. In literals, newlines are preserved:

Listing 94-4
```
1 |
2   \/ /| |\/| |
3   / / | |  | |__
```

Alternatively, strings can be written with the folded style, denoted by `>`, where each line break is replaced by a space:

Listing 94-5
```
1 >
2   This is a very long sentence
3   that spans several lines in the YAML
4   but which will be rendered as a string
5   without carriage returns.
```

Notice the two spaces before each line in the previous examples. They won't appear in the resulting PHP strings.

Numbers

Listing 94-6
```
1 # an integer
2 12
```

Listing 94-7

```
1 # an octal
2 014
```

Listing 94-8

```
1 # an hexadecimal
2 0xC
```

Listing 94-9

```
1 # a float
2 13.4
```

Listing 94-10

```
1 # an exponential number
2 1.2e+34
```

Listing 94-11

```
1 # infinity
2 .inf
```

Nulls

Nulls in YAML can be expressed with `null` or `~`.

Booleans

Booleans in YAML are expressed with `true` and `false`.

Dates

YAML uses the ISO-8601 standard to express dates:

Listing 94-12

```
1 2001-12-14t21:59:43.10-05:00
```

Listing 94-13

```
1 # simple date
2 2002-12-14
```

Collections

A YAML file is rarely used to describe a simple scalar. Most of the time, it describes a collection. A collection can be a sequence or a mapping of elements. Both sequences and mappings are converted to PHP arrays.

Sequences use a dash followed by a space:

Listing 94-14

```
1 - PHP
2 - Perl
3 - Python
```

The previous YAML file is equivalent to the following PHP code:

Listing 94-15

```
1 array('PHP', 'Perl', 'Python');
```

Mappings use a colon followed by a space (`: `) to mark each key/value pair:

Listing 94-16
```
1 PHP: 5.2
2 MySQL: 5.1
3 Apache: 2.2.20
```

which is equivalent to this PHP code:

Listing 94-17
```
1 array('PHP' => 5.2, 'MySQL' => 5.1, 'Apache' => '2.2.20');
```

In a mapping, a key can be any valid scalar.

The number of spaces between the colon and the value does not matter:

Listing 94-18
```
1 PHP:    5.2
2 MySQL:  5.1
3 Apache: 2.2.20
```

YAML uses indentation with one or more spaces to describe nested collections:

Listing 94-19
```
1 'symfony 1.0':
2   PHP:    5.0
3   Propel: 1.2
4 'symfony 1.2':
5   PHP:    5.2
6   Propel: 1.3
```

The above YAML is equivalent to the following PHP code:

Listing 94-20
```
1  array(
2      'symfony 1.0' => array(
3          'PHP'    => 5.0,
4          'Propel' => 1.2,
5      ),
6      'symfony 1.2' => array(
7          'PHP'    => 5.2,
8          'Propel' => 1.3,
9      ),
10 );
```

There is one important thing you need to remember when using indentation in a YAML file: *Indentation must be done with one or more spaces, but never with tabulators*.

You can nest sequences and mappings as you like:

Listing 94-21
```
1 'Chapter 1':
2   - Introduction
3   - Event Types
4 'Chapter 2':
```

```
5     - Introduction
6     - Helpers
```

YAML can also use flow styles for collections, using explicit indicators rather than indentation to denote scope.

A sequence can be written as a comma separated list within square brackets ([]):

Listing 94-22
```
1  [PHP, Perl, Python]
```

A mapping can be written as a comma separated list of key/values within curly braces ({}):

Listing 94-23
```
1  { PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

You can mix and match styles to achieve a better readability:

Listing 94-24
```
1  'Chapter 1': [Introduction, Event Types]
2  'Chapter 2': [Introduction, Helpers]
```

Listing 94-25
```
1  'symfony 1.0': { PHP: 5.0, Propel: 1.2 }
2  'symfony 1.2': { PHP: 5.2, Propel: 1.3 }
```

Comments

Comments can be added in YAML by prefixing them with a hash mark (#):

Listing 94-26
```
1  # Comment on a line
2  "symfony 1.0": { PHP: 5.0, Propel: 1.2 } # Comment at the end of a line
3  "symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Comments are simply ignored by the YAML parser and do not need to be indented according to the current level of nesting in a collection.