Symfony

The Components Book

Version: 3.0

generated on September 1, 2016

Contents at a Glance

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

```php
// 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
```
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\VersionStrategy\EmptyVersionStrategy;

$package = new Package(new EmptyVersionStrategy());

echo $package->getUrl('/image.png');
// 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
```
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\VersionStrategy\StaticVersionStrategy;

$package = new Package(new StaticVersionStrategy('v1'));

echo $package->getUrl('/image.png');
// 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:

---

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

Listing 2-4
```php
// put the 'version' word before the version value
$package = new Package(new StaticVersionStrategy('v1', '%s?version=%s'));

echo $package->getUrl('/image.png');
// result: /image.png?version=v1

// put the asset version before its path
$package = new Package(new StaticVersionStrategy('v1', '%2$s/%1$s'));

echo $package->getUrl('/image.png');
// 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
```php
use Symfony\Component\Asset\VersionStrategy\VersionStrategyInterface;

class DateVersionStrategy implements VersionStrategyInterface
{
    private $version;

    public function __construct()
    {
        $this->version = date('Ymd');
    }

    public function getVersion($path)
    {
        return $this->version;
    }

    public function applyVersion($path)
    {
        return sprintf('%s?v=%s', $path, $this->getVersion($path));
    }
}
```

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
```php
use Symfony\Component\Asset\PathPackage;
// ...

$package = new PathPackage('/static/images', new StaticVersionStrategy('v1'));

echo $package->getUrl('/logo.png');
// 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

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

```php
use Symfony\Component\Asset\PathPackage;
use Symfony\Component\Asset\Context\RequestStackContext;
// ...

$package = new PathPackage(
    '/static/images',
    new StaticVersionStrategy('v1'),
    new RequestStackContext($requestStack)
);

echo $package->getUrl('/logo.png');
// 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
```php
use Symfony\Component\Asset\UrlPackage;
// ...

$package = new UrlPackage(
    'http://static.example.com/images/',
    new StaticVersionStrategy('v1')
);

echo $package->getUrl('/logo.png');
// result: http://static.example.com/images/logo.png?v1
```

You can also pass a schema-agnostic URL:

Listing 2-9
```php
use Symfony\Component\Asset\UrlPackage;
// ...

$package = new UrlPackage(
    '//static.example.com/images/',
    new StaticVersionStrategy('v1')
);

echo $package->getUrl('/logo.png');
// 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
```php
use Symfony\Component\Asset\UrlPackage;
// ...

$urls = array(
    '//static1.example.com/images/',
    '//static2.example.com/images/',
);
$package = new UrlPackage($urls, new StaticVersionStrategy('v1'));
```

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

```
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
```
1  use Symfony\Component\Asset\Package;
2  use Symfony\Component\Asset\PathPackage;
3  use Symfony\Component\Asset\UrlPackage;
4  use Symfony\Component\Asset\Packages;
5  // ...
6
7  $versionStrategy = new StaticVersionStrategy('v1');
8
9  $defaultPackage = new Package($versionStrategy);
10
11 $namedPackages = array(
12     'img' => new UrlPackage('http://img.example.com/', $versionStrategy),
13     'doc' => new PathPackage('/somewhere/deep/for/documents', $versionStrategy),
14 );
15
16 $packages = new Packages($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:

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

Listing 2-13
```php
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
```

Learn more

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
```
1  namespace Acme;
2
3  use Symfony\Component\BrowserKit\Client as BaseClient;
4  use Symfony\Component\BrowserKit\Response;
5
6  class Client extends BaseClient
7  {
8      protected function doRequest($request)
```

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

```
    {
        // ... convert request into a response

        return new Response($content, $status, $headers);
    }
}
```

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
```
use Acme\Client;

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

// make a real request to an external site
$client = new Client();
$crawler = $client->request('GET', 'https://github.com/login');

// select the form and fill in some values
```

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

```php
$form = $crawler->selectButton('Log in')->form();
$form['login'] = 'symfonyfan';
$form['password'] = 'anypass';

// submit that form
$crawler = $client->submit($form);
```

Cookies

Retrieving Cookies

The `Client` implementation 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

```php
use Acme\Client;

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

// Get the cookie Jar
$cookieJar = $client->getCookieJar();

// Get a cookie by name
$cookie = $cookieJar->get('name_of_the_cookie');

// Get cookie data
$name       = $cookie->getName();
$value      = $cookie->getValue();
$raw        = $cookie->getRawValue();
$secure     = $cookie->isSecure();
$isHttpOnly = $cookie->isHttpOnly();
$isExpired  = $cookie->isExpired();
$expires    = $cookie->getExpiresTime();
$path       = $cookie->getPath();
$domain     = $cookie->getDomain();
```

These methods only return cookies that have not expired.

Looping Through Cookies

Listing 3-6

```php
use Acme\Client;

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

// Get the cookie Jar
$cookieJar = $client->getCookieJar();

// Get array with all cookies
$cookies = $cookieJar->all();
foreach ($cookies as $cookie) {
    // ...
}
```

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

```
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();
```

Learn more

- *Testing*

- *The CssSelector Component*
- *The DomCrawler Component*

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*[3] 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*[4]);
- Use the official Git repository (*https://github.com/symfony/class-loader*).

1. `http://php.net/manual/en/language.oop5.autoload.php`
2. `http://www.php-fig.org/psr/psr-0/`
3. `http://www.php-fig.org/psr/psr-4/`
4. `https://packagist.org/packages/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.

Learn More

- The PSR-0 Class Loader
- Cache a Class Loader
- The Class Map Generator
- Debugging a Class Loader
- MapClassLoader
- The PSR-4 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
```php
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();
```

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

Listing 5-2
```php
// register a single namespaces
$loader->addPrefix('Symfony', __DIR__.'/vendor/symfony/symfony/src');

```

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
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

```php
// 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

```php
$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.

6. http://pear.php.net/manual/en/standards.naming.php

Chapter 6
Cache a Class Loader

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 6-1

```php
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();

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

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

XcacheClassLoader

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

Listing 6-2

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

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

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

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

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

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

Chapter 7
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 7-1

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

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 7-4
```
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 7-5
```
<?php return array (
'Acme\\Foo' => '/var/www/library/foo/Bar.php',
'Acme\\Foo\\Bar' => '/var/www/library/foo/bar/Foo.php',
'Acme\\Bar\\Baz' => '/var/www/library/bar/baz/Boo.php',
'Acme\\Bar' => '/var/www/library/bar/Foo.php',
);
```

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

Listing 7-6
```
use Symfony\Component\ClassLoader\MapClassLoader;

$mapping = include __DIR__.'/class_map.php';
$loader = new MapClassLoader($mapping);
$loader->register();

// you can now use the classes:
use Acme\Foo;

$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

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 7-7

```php
use Symfony\Component\ClassLoader\ClassMapGenerator;

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

7. https://getcomposer.org

Chapter 8
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 9
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 9-1

```php
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 10
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 10-1

```
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`:

Listing 10-2

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

```php
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 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.

Learn More

- Caching based on Resources
- Defining and Processing Configuration Values
- Loading Resources
- How to Create Friendly Configuration for a Bundle
- How to Load Service Configuration inside a Bundle
- How to Simplify Configuration of multiple Bundles

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

Chapter 12
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 12-1

```php
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);
    }

    // the code for the UserMatcher is generated elsewhere
    $code = ...;

    $userMatcherCache->write($code, $resources);
```

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

```
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 13

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 13-1

```
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 13-2
```php
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 13-3
```php
$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:

- 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)

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

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 13-4
```
$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 13-5
```
$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 13-6
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')->end()
                ->scalarNode('host')->end()
                ->scalarNode('username')->end()
                ->scalarNode('password')->end()
            ->end()
        ->end()
    ->end()
;
```

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

Listing 13-7
```
$rootNode
    ->children()
        ->arrayNode('connections')
            ->prototype('array')
```

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

```
                ->children()
                    ->scalarNode('driver')->end()
                    ->scalarNode('host')->end()
                    ->scalarNode('username')->end()
                    ->scalarNode('password')->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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 13-8
```
$node
    ->fixXmlConfig('driver')
    ->children()
        ->arrayNode('drivers')
            ->prototype('scalar')->end()
        ->end()
    ->end()
;
```

When using the following YAML configuration:

Listing 13-9
```
drivers: ['mysql', 'sqlite']
```

Or the following XML configuration:

Listing 13-10
```
<driver>mysql</driver>
<driver>sqlite</driver>
```

The processed configuration is:

Listing 13-11
```
Array(
    [0] => 'mysql'
    [1] => 'sqlite'
)
```

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

Listing 13-12
```
$node
    ->fixXmlConfig('connection')
    ->children()
        ->arrayNode('connections')
            ->prototype('array')
                ->children()
                    ->scalarNode('table')->end()
                    ->scalarNode('user')->end()
                    ->scalarNode('password')->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

When using the following YAML configuration:

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

Or the following XML configuration:

Listing 13-14
```
<connection table="symfony" user="root" password="null" />
<connection table="foo" user="root" password="pa$$" />
```

The processed configuration is:

Listing 13-15
```
Array(
    [0] => Array(
        [table] => 'symfony'
        [user] => 'root'
        [password] => null
    )
    [1] => Array(
        [table] => 'foo'
        [user] => 'root'
        [password] => 'pa$$'
    )
)
```

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

Listing 13-16
```
connections:
    sf_connection:
        table: symfony
        user: root
        password: ~
    default:
        table: foo
        user: root
        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 13-17
```
$node
    ->fixXmlConfig('connection')
    ->children()
        ->arrayNode('connections')
            ->useAttributeAsKey('name')
            ->prototype('array')
                ->children()
                    ->scalarNode('table')->end()
                    ->scalarNode('user')->end()
                    ->scalarNode('password')->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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 13-18
```
<connection name="sf_connection"
    table="symfony" user="root" password="null" />
<connection name="default"
    table="foo" user="root" password="pa$$" />
```

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

Listing 13-19
```
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 13-20
```
$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()
            ->end()
        ->end()
        ->arrayNode('settings')
            ->addDefaultsIfNotSet()
            ->children()
                ->scalarNode('name')
                    ->isRequired()
                    ->cannotBeEmpty()
                    ->defaultValue('value')
                ->end()
            ->end()
        ->end()
    ->end()
;
```

Documenting the Option

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

Listing 13-21
```
$rootNode
    ->children()
        ->integerNode('entries_per_page')
            ->info('This value is only used for the search results page.')
            ->defaultValue(25)
        ->end()
    ->end()
;
```

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 13-22
```
# This value is only used for the search results page.
entries_per_page:     25
```

and in XML:

Listing 13-23
```
<!-- entries-per-page: This value is only used for the search results page. -->
<config entries-per-page="25" />
```

6. http://api.symfony.com/3.0/Symfony/Component/Config/Definition/Builder/NodeDefinition.html#method_info

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 13-24
```
$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 13-25
```
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()
```

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

```
                    ->scalarNode('password')->end()
                    ->booleanNode('memory')
                        ->defaultFalse()
                    ->end()
                ->end()
                ->append($this->addParametersNode())
            ->end()
        ->end()
    ;

    return $treeBuilder;
}

public function addParametersNode()
{
    $builder = new TreeBuilder();
    $node = $builder->root('parameters');

    $node
        ->isRequired()
        ->requiresAtLeastOneElement()
        ->useAttributeAsKey('name')
        ->prototype('array')
            ->children()
                ->scalarNode('value')->isRequired()->end()
            ->end()
        ->end()
    ;

    return $node;
}
```

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 13-26

```
database:
    connection:
        driver:               ~ # Required
        host:                 localhost
        username:             ~
        password:             ~
        memory:               false
        parameters:           # Required

            # Prototype
            name:
                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 13-27
```yaml
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 13-28
```xml
<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 13-29
```php
$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 13-30
```php
$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 13-31
```xml
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 13-32
```xml
<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 13-33
```yaml
connection:
    name:     my_mysql_connection
    host:     localhost
    driver:   mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 13-34

```
connection: my_mysql_connection
```

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

Listing 13-35
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
                ->ifString()
                ->then(function ($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 13-36
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->validate()
                    ->ifNotInArray(array('mysql', 'sqlite', 'mssql'))
                        ->thenInvalid('Invalid database driver %s')
                    ->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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()`

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

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 13-37
```php
use Symfony\Component\Yaml\Yaml;
use Symfony\Component\Config\Definition\Processor;
use Acme\DatabaseConfiguration;

$config1 = Yaml::parse(
    file_get_contents(__DIR__.'/src/Matthias/config/config.yml')
);
$config2 = Yaml::parse(
    file_get_contents(__DIR__.'/src/Matthias/config/config_extra.yml')
);

$configs = array($config1, $config2);

$processor = new Processor();
$configuration = new DatabaseConfiguration();
$processedConfiguration = $processor->processConfiguration(
    $configuration,
    $configs
);
```

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 14
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 14-1
```
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.

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:

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

Listing 14-2
```php
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 14-3
```php
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');
/*
The YamlUserLoader will be used to load this resource,
since it supports files with a "yml" extension
*/
```

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