Symfony

The Components Book

Version: 4.0

generated on January 20, 2018

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/');

// ...
```

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

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

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

// Relative path
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
```php
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\VersionStrategy\StaticVersionStrategy;

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

```

1. https://packagist.org/packages/symfony/asset
2. http://api.symfony.com/4.0/Symfony/Component/Asset/PackageInterface.html
3. **http://api.symfony.com/4.0/Symfony/Component/Asset/PackageInterface.html#method_getVersion**
4. **http://api.symfony.com/4.0/Symfony/Component/Asset/PackageInterface.html#method_getUrl**
5. http://api.symfony.com/4.0/Symfony/Component/Asset/VersionStrategy/EmptyVersionStrategy.html
6. http://api.symfony.com/4.0/Symfony/Component/Asset/VersionStrategy/StaticVersionStrategy.html

```php
// Absolute path
echo $package->getUrl('/image.png');
// result: /image.png?v1

// Relative path
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:

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

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

7. http://api.symfony.com/4.0/Symfony/Component/Asset/VersionStrategy/VersionStrategyInterface.html
8. http://api.symfony.com/4.0/Symfony/Component/Asset/Package.html
9. http://api.symfony.com/4.0/Symfony/Component/Asset/PathPackage.html

```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

// Base path is ignored when using absolute paths
echo $package->getUrl('/logo.png');
// result: /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

```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

// Both "base path" and "base url" are ignored when using absolute path for asset
echo $package->getUrl('/logo.png');
// result: /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;
// ...

```

10. http://api.symfony.com/4.0/Symfony/Component/Asset/UrlPackage.html

```php
$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'));

echo $package->getUrl('/logo.png');
// result: http://static1.example.com/images/logo.png?v1
echo $package->getUrl('/icon.png');
// 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
```php
use Symfony\Component\Asset\UrlPackage;
use Symfony\Component\Asset\Context\RequestStackContext;
// ...

$package = new UrlPackage(
    array('http://example.com/', 'https://example.com/'),
    new StaticVersionStrategy('v1'),
    new RequestStackContext($requestStack)
);

echo $package->getUrl('/logo.png');
// assuming the RequestStackContext says that we are on a secure host
// 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:

11. http://api.symfony.com/4.0/Symfony/Component/Asset/Packages.html

Listing 2-12
```php
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
```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.

The BrowserKit component can only make internal requests to your application. If you need to make requests to external sites and applications, consider using *Goutte*[1], a simple web scraper based on Symfony Components.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/browser-kit` on *Packagist*[2]);
- 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()`*[3] method. This method accepts a request and should return a response:

Listing 3-1

1. `https://github.com/FriendsOfPHP/Goutte`
2. `https://packagist.org/packages/symfony/browser-kit`
3. `http://api.symfony.com/4.0/Symfony/Component/BrowserKit/Client.html#method_doRequest`

```php
namespace Acme;

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

class Client extends BaseClient
{
    protected function doRequest($request)
    {
        // ... 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*[4]. For an implementation based on `HttpKernelInterface`, have a look at the *`Client`*[5] provided by the *HttpKernel component*.

Making Requests

Use the *`request()`*[6] method to make HTTP requests. The first two arguments are the HTTP method and the requested URL:

Listing 3-2

```php
use Acme\Client;

$client = new Client();
$crawler = $client->request('GET', '/');
```

The value returned by the `request()` method is an instance of the *`Crawler`*[7] 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

```php
use Acme\Client;

$client = new Client();
$crawler = $client->request('GET', '/product/123');
$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):

4. https://github.com/FriendsOfPHP/Goutte
5. http://api.symfony.com/4.0/Symfony/Component/HttpKernel/Client.html
6. http://api.symfony.com/4.0/Symfony/Component/BrowserKit/Client.html#method_request
7. http://api.symfony.com/4.0/Symfony/Component/DomCrawler/Crawler.html

Listing 3-4

```php
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
$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`*[8], 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', '/');

// 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();
$sameSite   = $cookie->getSameSite();
```

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', '/');

```

8. http://api.symfony.com/4.0/Symfony/Component/BrowserKit/CookieJar.html

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

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

// Get all values
$values = $cookieJar->allValues('http://symfony.com');
foreach ($values as $value) {
    // ...
}

// Get all raw values
$rawValues = $cookieJar->allRawValues('http://symfony.com');
foreach ($rawValues as $rawValue) {
    // ...
}
```

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
```php
use Acme\Client;

// create cookies and add to cookie jar
$cookieJar = new Cookie('flavor', 'chocolate', strtotime('+1 day'));

// create a client and set the cookies
$client = new Client(array(), null, $cookieJar);
// ...
```

History

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

Listing 3-8
```php
use Acme\Client;

$client = new Client();
$client->request('GET', '/');

// select and click on a link
$link = $crawler->selectLink('Documentation')->link();
$client->click($link);

// go back to home page
$crawler = $client->back();

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

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

Listing 3-9
```php
use Acme\Client;

$client = new Client();
$client->request('GET', '/');

// reset the client (history and cookies are cleared too)
$client->restart();
```

Learn more

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

Chapter 4
The Cache Component

> The Cache component provides an extended *PSR-6*[1] implementation as well as a *PSR-16*[2] "Simple Cache" implementation for adding cache to your applications. It is designed to have a low overhead and it ships with ready to use adapters for the most common caching backends.

Installation

You can install the component in 2 different ways:

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

Cache (PSR-6) Versus Simple Cache (PSR-16)

This component includes *two* different approaches to caching:

PSR-6 Caching:
: A fully-featured cache system, which includes cache "pools", more advanced cache "items", and cache tagging for invalidation.

PSR-16 Simple Caching:
: A simple way to store, fetch and remove items from a cache.

Both methods support the *same* cache adapters and will give you very similar performance.

The component also contains adapters to convert between PSR-6 and PSR-16 caches. See *Adapters For Interoperability between PSR-6 and PSR-16 Cache*.

1. `http://www.php-fig.org/psr/psr-6/`
2. `http://www.php-fig.org/psr/psr-16/`
3. `https://packagist.org/packages/symfony/cache`

Simple Caching (PSR-16)

This part of the component is an implementation of *PSR-16*[4], which means that its basic API is the same as defined in the standard. First, create a cache object from one of the built-in cache classes. For example, to create a filesystem-based cache, instantiate *`FilesystemCache`*[5]:

Listing 4-1
```php
use Symfony\Component\Cache\Simple\FilesystemCache;

$cache = new FilesystemCache();
```

Now you can create, retrieve, update and delete items using this object:

Listing 4-2
```php
// save a new item in the cache
$cache->set('stats.num_products', 4711);

// or set it with a custom ttl
// $cache->set('stats.num_products', 4711, 3600);

// retrieve the cache item
if (!$cache->has('stats.num_products')) {
    // ... item does not exists in the cache
}

// retrieve the value stored by the item
$numProducts = $cache->get('stats.num_products');

// or specify a default value, if the key doesn't exist
// $numProducts = $cache->get('stats.num_products', 100);

// remove the cache key
$cache->delete('stats.num_products');

// clear *all* cache keys
$cache->clear();
```

You can also work with multiple items at once:

Listing 4-3
```php
$cache->setMultiple(array(
    'stats.num_products' => 4711,
    'stats.num_users' => 1356,
));

$stats = $cache->getMultiple(array(
    'stats.num_products',
    'stats.num_users',
));

$cache->deleteMultiple(array(
    'stats.num_products',
    'stats.num_users',
));
```

Available Simple Cache (PSR-16) Classes

The following cache adapters are available:

To find out more about each of these classes, you can read the *PSR-6 Cache Pool* page. These "Simple" (PSR-16) cache classes aren't identical to the PSR-6 Adapters on that page, but each share constructor arguments and use-cases.

- *ApcuCache*[6]

4. http://www.php-fig.org/psr/psr-16/
5. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/FilesystemCache.html

- *ArrayCache*[7]
- *ChainCache*[8]
- *DoctrineCache*[9]
- *FilesystemCache*[10]
- *MemcachedCache*[11]
- *NullCache*[12]
- *PdoCache*[13]
- *PhpArrayCache*[14]
- *PhpFilesCache*[15]
- *RedisCache*[16]
- *TraceableCache*[17]

More Advanced Caching (PSR-6)

To use the more-advanced, PSR-6 Caching abilities, you'll need to learn its key concepts:

Item
: A single unit of information stored as a key/value pair, where the key is the unique identifier of the information and the value is its contents;

Pool
: A logical repository of cache items. All cache operations (saving items, looking for items, etc.) are performed through the pool. Applications can define as many pools as needed.

Adapter
: It implements the actual caching mechanism to store the information in the filesystem, in a database, etc. The component provides several ready to use adapters for common caching backends (Redis, APCu, Doctrine, PDO, etc.)

Basic Usage (PSR-6)

This part of the component is an implementation of *PSR-6*[18], which means that its basic API is the same as defined in the standard. Before starting to cache information, create the cache pool using any of the built-in adapters. For example, to create a filesystem-based cache, instantiate *FilesystemAdapter*[19]:

Listing 4-4
```php
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter();
```

Now you can create, retrieve, update and delete items using this cache pool:

Listing 4-5
```php
// create a new item by trying to get it from the cache
$numProducts = $cache->getItem('stats.num_products');

// assign a value to the item and save it
```

6. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/ApcuCache.html
7. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/ArrayCache.html
8. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/ChainCache.html
9. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/DoctrineCache.html
10. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/FilesystemCache.html
11. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/MemcachedCache.html
12. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/NullCache.html
13. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/PdoCache.html
14. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/PhpArrayCache.html
15. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/PhpFilesCache.html
16. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/RedisCache.html
17. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/TraceableCache.html
18. http://www.php-fig.org/psr/psr-6/
19. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/FilesystemAdapter.html

```
$numProducts->set(4711);
$cache->save($numProducts);

// retrieve the cache item
$numProducts = $cache->getItem('stats.num_products');
if (!$numProducts->isHit()) {
    // ... item does not exists in the cache
}
// retrieve the value stored by the item
$total = $numProducts->get();

// remove the cache item
$cache->deleteItem('stats.num_products');
```

For a list of all of the supported adapters, see *Cache Pools and Supported Adapters*.

Advanced Usage (PSR-6)

- Cache Invalidation
- Cache Items
- Cache Pools and Supported Adapters
- Adapters For Interoperability between PSR-6 and PSR-16 Cache

Chapter 5
Cache Invalidation

Cache invalidation is the process of removing all cached items related to a change in the state of your model. The most basic kind of invalidation is direct items deletion. But when the state of a primary resource has spread across several cached items, keeping them in sync can be difficult.

The Symfony Cache component provides two mechanisms to help solving this problem:

- Tags based invalidation for managing data dependencies;
- Expiration based invalidation for time related dependencies.

Using Cache Tags

To benefit from tags based invalidation, you need to attach the proper tags to each cached item. Each tag is a plain string identifier that you can use at any time to trigger the removal of all items associated with this tag.

To attach tags to cached items, you need to use the *`tag()`*[1] method that is implemented by cache items, as returned by cache adapters:

Listing 5-1
```php
$item = $cache->getItem('cache_key');
// ...
// add one or more tags
$item->tag('tag_1');
$item->tag(array('tag_2', 'tag_3'));
$cache->save($item);
```

If `$cache` implements *`TagAwareAdapterInterface`*[2], you can invalidate the cached items by calling *`invalidateTags()`*[3]:

Listing 5-2
```php
// invalidate all items related to `tag_1` or `tag_3`
$cache->invalidateTags(array('tag_1', 'tag_3'));

```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/CacheItem.html#method_tag
2. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapterInterface.html
3. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapterInterface.html#method_invalidateTags

```
// if you know the cache key, you can also delete the item directly
$cache->deleteItem('cache_key');
```

Using tags invalidation is very useful when tracking cache keys becomes difficult.

Tag Aware Adapters

To store tags, you need to wrap a cache adapter with the *TagAwareAdapter*[4] class or implement *TagAwareAdapterInterface*[5] and its only *invalidateTags()*[6] method.

The *TagAwareAdapter*[7] class implements instantaneous invalidation (time complexity is `O(N)` where `N` is the number of invalidated tags). It needs one or two cache adapters: the first required one is used to store cached items; the second optional one is used to store tags and their invalidation version number (conceptually similar to their latest invalidation date). When only one adapter is used, items and tags are all stored in the same place. By using two adapters, you can e.g. store some big cached items on the filesystem or in the database and keep tags in a Redis database to sync all your fronts and have very fast invalidation checks:

Listing 5-3
```
use Symfony\Component\Cache\Adapter\TagAwareAdapter;
use Symfony\Component\Cache\Adapter\FilesystemAdapter;
use Symfony\Component\Cache\Adapter\RedisAdapter;

$cache = new TagAwareAdapter(
    // Adapter for cached items
    new FilesystemAdapter(),
    // Adapter for tags
    new RedisAdapter('redis://localhost')
);
```

Using Cache Expiration

If your data is valid only for a limited period of time, you can specify their lifetime or their expiration date with the PSR-6 interface, as explained in the *Cache Items* article.

4. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapter.html
5. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapterInterface.html
6. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapterInterface.html#method_invalidateTags
7. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/TagAwareAdapter.html

Chapter 6
Cache Items

Cache items are the information units stored in the cache as a key/value pair. In the Cache component they are represented by the *CacheItem*[1] class.

Cache Item Keys and Values

The **key** of a cache item is a plain string which acts as its identifier, so it must be unique for each cache pool. You can freely choose the keys, but they should only contain letters (A-Z, a-z), numbers (0-9) and the `_` and `.` symbols. Other common symbols (such as `{`, `}`, `(`, `)`, `/`, `\` and `@`) are reserved by the PSR-6 standard for future uses.

The **value** of a cache item can be any data represented by a type which is serializable by PHP, such as basic types (string, integer, float, boolean, null), arrays and objects.

Creating Cache Items

Cache items are created with the `getItem($key)` method of the cache pool. The argument is the key of the item:

Listing 6-1
```
// $cache pool object was created before
$numProducts = $cache->getItem('stats.num_products');
```

Then, use the *set()*[2] method to set the data stored in the cache item:

Listing 6-2
```
// storing a simple integer
$numProducts->set(4711);
$cache->save($numProducts);

// storing an array
$numProducts->set(array(
    'category1' => 4711,
    'category2' => 2387,
```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/CacheItem.html
2. http://api.symfony.com/4.0/Psr/Cache/CacheItemInterface.html#method_set

```
 9  ));
10  $cache->save($numProducts);
```

The key and the value of any given cache item can be obtained with the corresponding *getter* methods:

Listing 6-3
```
$cacheItem = $cache->getItem('exchange_rate');
// ...
$key = $cacheItem->getKey();
$value = $cacheItem->get();
```

Cache Item Expiration

By default cache items are stored permanently. In practice, this "permanent storage" can vary greatly depending on the type of cache being used, as explained in the *Cache Pools and Supported Adapters* article.

However, in some applications it's common to use cache items with a shorter lifespan. Consider for example an application which caches the latest news just for one minute. In those cases, use the `expiresAfter()` method to set the number of seconds to cache the item:

Listing 6-4
```
1  $latestNews = $cache->getItem('latest_news');
2  $latestNews->expiresAfter(60);  // 60 seconds = 1 minute
3
4  // this method also accepts \DateInterval instances
5  $latestNews->expiresAfter(DateInterval::createFromDateString('1 hour'));
```

Cache items define another related method called `expiresAt()` to set the exact date and time when the item will expire:

Listing 6-5
```
$mostPopularNews = $cache->getItem('popular_news');
$mostPopularNews->expiresAt(new \DateTime('tomorrow'));
```

Cache Item Hits and Misses

Using a cache mechanism is important to improve the application performance, but it should not be required to make the application work. In fact, the PSR-6 standard states that caching errors should not result in application failures.

In practice this means that the `getItem()` method always returns an object which implements the `Psr\Cache\CacheItemInterface` interface, even when the cache item doesn't exist. Therefore, you don't have to deal with `null` return values and you can safely store in the cache values such as `false` and `null`.

In order to decide if the returned object is correct or not, caches use the concept of hits and misses:

- **Cache Hits** occur when the requested item is found in the cache, its value is not corrupted or invalid and it hasn't expired;
- **Cache Misses** are the opposite of hits, so they occur when the item is not found in the cache, its value is corrupted or invalid for any reason or the item has expired.

Cache item objects define a boolean `isHit()` method which returns `true` for cache hits:

Listing 6-6
```
1  $latestNews = $cache->getItem('latest_news');
2
3  if (!$latestNews->isHit()) {
4      // do some heavy computation
5      $news = ...;
6      $cache->save($latestNews->set($news));
7  } else {
```

```
8       $news = $latestNews->get();
9   }
```

Chapter 7
Cache Pools and Supported Adapters

Cache Pools are the logical repositories of cache items. They perform all the common operations on items, such as saving them or looking for them. Cache pools are independent from the actual cache implementation. Therefore, applications can keep using the same cache pool even if the underlying cache mechanism changes from a file system based cache to a Redis or database based cache.

Creating Cache Pools

Cache Pools are created through the **cache adapters**, which are classes that implement *`AdapterInterface`*[1]. This component provides several adapters ready to use in your applications.

- APCu Cache Adapter
- Array Cache Adapter
- Chain Cache Adapter
- Doctrine Cache Adapter
- Filesystem Cache Adapter
- Memcached Cache Adapter
- PDO & Doctrine DBAL Cache Adapter
- Php Array Cache Adapter
- Proxy Cache Adapter
- Redis Cache Adapter

Looking for Cache Items

Cache Pools define three methods to look for cache items. The most common method is `getItem($key)`, which returns the cache item identified by the given key:

Listing 7-1
```
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter('app.cache');
$latestNews = $cache->getItem('latest_news');
```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/AdapterInterface.html

If no item is defined for the given key, the method doesn't return a `null` value but an empty object which implements the *CacheItem*[2] class.

If you need to fetch several cache items simultaneously, use instead the `getItems(array($key1, $key2, ...))` method:

Listing 7-2
```
// ...
$stocks = $cache->getItems(array('AAPL', 'FB', 'GOOGL', 'MSFT'));
```

Again, if any of the keys doesn't represent a valid cache item, you won't get a `null` value but an empty `CacheItem` object.

The last method related to fetching cache items is `hasItem($key)`, which returns `true` if there is a cache item identified by the given key:

Listing 7-3
```
// ...
$hasBadges = $cache->hasItem('user_'.$userId.'_badges');
```

Saving Cache Items

The most common method to save cache items is *save()*[3], which stores the item in the cache immediately (it returns `true` if the item was saved or `false` if some error occurred):

Listing 7-4
```
// ...
$userFriends = $cache->getItem('user_'.$userId.'_friends');
$userFriends->set($user->getFriends());
$isSaved = $cache->save($userFriends);
```

Sometimes you may prefer to not save the objects immediately in order to increase the application performance. In those cases, use the *saveDeferred()*[4] method to mark cache items as "ready to be persisted" and then call to *commit()*[5] method when you are ready to persist them all:

Listing 7-5
```
// ...
$isQueued = $cache->saveDeferred($userFriends);
// ...
$isQueued = $cache->saveDeferred($userPreferences);
// ...
$isQueued = $cache->saveDeferred($userRecentProducts);
// ...
$isSaved = $cache->commit();
```

The `saveDeferred()` method returns `true` when the cache item has been successfully added to the "persist queue" and `false` otherwise. The `commit()` method returns `true` when all the pending items are successfully saved or `false` otherwise.

Removing Cache Items

Cache Pools include methods to delete a cache item, some of them or all of them. The most common is *deleteItem()*[6], which deletes the cache item identified by the given key (it returns `true` when the item is successfully deleted or doesn't exist and `false` otherwise):

2. http://api.symfony.com/4.0/Symfony/Component/Cache/CacheItem.html
3. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_save
4. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_saveDeferred
5. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_commit
6. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_deleteItem

Listing 7-6
```
// ...
$isDeleted = $cache->deleteItem('user_'.$userId);
```

Use the *`deleteItems()`*[7] method to delete several cache items simultaneously (it returns `true` only if all the items have been deleted, even when any or some of them don't exist):

Listing 7-7
```
// ...
$areDeleted = $cache->deleteItems(array('category1', 'category2'));
```

Finally, to remove all the cache items stored in the pool, use the *`clear()`*[8] method (which returns `true` when all items are successfully deleted):

Listing 7-8
```
// ...
$cacheIsEmpty = $cache->clear();
```

If the Cache component is used inside a Symfony application, you can remove all the items of a given cache pool with the following command:

Listing 7-9
```
$ php bin/console cache:pool:clear <cache-pool-name>

# clears the "cache.app" pool
$ php bin/console cache:pool:clear cache.app

# clears the "cache.validation" and "cache.app" pool
$ php bin/console cache:pool:clear cache.validation cache.app
```

7. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_deleteItems
8. http://api.symfony.com/4.0/Psr/Cache/CacheItemPoolInterface.html#method_clear

Chapter 8
APCu Cache Adapter

This adapter is a high-performance, shared memory cache. It can increase the application performance very significantly because the cache contents are stored in the shared memory of your server, a component that is much faster than others, such as the filesystem.

Requirement: The *APCu extension*[1] must be installed and active to use this adapter.

This adapter can be provided an optional namespace string as its first parameter, a default cache lifetime as its second parameter, and a version string as its third parameter:

Listing 8-1
```php
use Symfony\Component\Cache\Adapter\ApcuAdapter;

$cache = new ApcuAdapter(

    // a string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the APCu memory is cleared)
    $defaultLifetime = 0,

    // when set, all keys prefixed by $namespace can be invalidated by changing
    // this $version string
    $version = null
);
```

It is *not* recommended to use this adapter when performing a large number of write and delete operations, as these operations result in fragmentation of the APCu memory, resulting in *significantly* degraded performance.

1. https://pecl.php.net/package/APCu

Note that this adapter's CRUD operations are specific to the PHP SAPI it is running under. This means adding a cache item using the CLI will not result in the item appearing under FPM. Likewise, deletion of an item using CGI will not result in the item being deleted under the CLI.

Chapter 9
Array Cache Adapter

Generally, this adapter is useful for testing purposes, as its contents are stored in memory and not persisted outside the running PHP process in any way. It can also be useful while warming up caches, due to the *`getValues()`*[1] method.

This adapter can be passed a default cache lifetime as its first parameter, and a boolean that toggles serialization as its second parameter:

Listing 9-1

```php
use Symfony\Component\Cache\Adapter\ArrayAdapter;

$cache = new ArrayAdapter(

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the current PHP process finishes)
    $defaultLifetime = 0,

    // if ``true``, the values saved in the cache are serialized before storing them
    $storeSerialized = true
);
```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/ArrayAdapter.html#method_getValues

Chapter 10
Chain Cache Adapter

This adapter allows to combine any number of the other available cache adapters. Cache items are fetched from the first adapter which contains them and cache items are saved in all the given adapters. This offers a simple way of creating a layered cache.

This adapter expects an array of adapters as its first parameter, and optionally a maximum cache lifetime as its second parameter:

Listing 10-1
```php
use Symfony\Component\Cache\Adapter\ApcuAdapter;

$cache = new ChainAdapter(array(

    // The ordered list of adapters used to fetch cached items
    array $adapters,

    // The max lifetime of items propagated from lower adapters to upper ones
    $maxLifetime = 0
));
```

When an item is not found in the first adapter but is found in the next ones, this adapter ensures that the fetched item is saved in all the adapters where it was previously missing.

The following example shows how to create a chain adapter instance using the fastest and slowest storage engines, *`ApcuAdapter`*[1] and *`FilesystemAdapter`*[2], respectfully:

Listing 10-2
```php
use Symfony\Component\Cache\Adapter\ApcuAdapter;
use Symfony\Component\Cache\Adapter\ChainAdapter;
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new ChainAdapter(array(
    new ApcuAdapter(),
    new FilesystemAdapter(),
));
```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/ApcuAdapter.html
2. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/FilesystemAdapter.html

Chapter 11
Doctrine Cache Adapter

This adapter wraps any class extending the *Doctrine Cache*[1] abstract provider, allowing you to use these providers in your application as if they were Symfony Cache adapters.

This adapter expects a `\Doctrine\Common\Cache\CacheProvider` instance as its first parameter, and optionally a namespace and default cache lifetime as its second and third parameters:

Listing 11-1
```php
use Doctrine\Common\Cache\CacheProvider;
use Doctrine\Common\Cache\SQLite3Cache;
use Symfony\Component\Cache\Adapter\DoctrineAdapter;

$provider = new SQLite3Cache(new \SQLite3(__DIR__.'/cache/data.sqlite'), 'youTableName');

$symfonyCache = new DoctrineAdapter(

    // a cache provider instance
    CacheProvider $provider,

    // a string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the database table is truncated or its rows are otherwise deleted)
    $defaultLifetime = 0
);
```

1. https://github.com/doctrine/cache

Chapter 12
Filesystem Cache Adapter

This adapter is useful when you want to improve the application performance but can't install tools like APCu or Redis on the server. This adapter stores the contents as regular files in a set of directories on the local file system.

This adapter can optionally be provided a namespace, default cache lifetime, and directory path, as its first, second, and third parameters:

Listing 12-1
```php
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter(

    // a string used as the subdirectory of the root cache directory, where cache
    // items will be stored
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the files are deleted)
    $defaultLifetime = 0,

    // the main cache directory (the application needs read-write permissions on it)
    // if none is specified, a directory is created inside the system temporary directory
    $directory = null
);
```

This adapter is generally the *slowest* due to the overhead of file IO. If throughput is paramount, the in-memory adapters (such as APCu, Memcached, and Redis) or the database adapters (such as Doctrine and PDO & Doctrine) are recommended.

Chapter 13
Memcached Cache Adapter

This adapter stores the values in-memory using one (or more) *Memcached server*[1] instances. Unlike the APCu adapter, and similarly to the Redis adapter, it is not limited to the current server's shared memory; you can store contents independent of your PHP environment. The ability to utilize a cluster of servers to provide redundancy and/or fail-over is also available.

Requirements: The *Memcached PHP extension*[2] as well as a *Memcached server*[3] must be installed, active, and running to use this adapter. Version `2.2` or greater of the *Memcached PHP extension*[4] is required for this adapter.

This adapter expects a *Memcached*[5] instance to be passed as the first parameter. A namespace and default cache lifetime can optionally be passed as the second and third parameters:

Listing 13-1
```php
use Symfony\Component\Cache\Adapter\MemcachedAdapter;

$cache = new MemcachedAdapter(
    // the client object that sets options and adds the server instance(s)
    \Memcached $client,

    // a string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until MemcachedAdapter::clear() is invoked or the server(s) are restarted)
    $defaultLifetime = 0
);
```

1. https://memcached.org/
2. http://php.net/manual/en/book.memcached.php
3. https://memcached.org/
4. http://php.net/manual/en/book.memcached.php
5. http://php.net/manual/en/class.memcached.php

Configure the Connection

The *createConnection()*[6] helper method allows creating and configuring a *Memcached*[7] class instance using a *Data Source Name (DSN)*[8] or an array of DSNs:

Listing 13-2
```php
use Symfony\Component\Cache\Adapter\MemcachedAdapter;

// pass a single DSN string to register a single server with the client
$client = MemcachedAdapter::createConnection(
    'memcached://localhost'
);

// pass an array of DSN strings to register multiple servers with the client
$client = MemcachedAdapter::createConnection(array(
    'memcached://10.0.0.100',
    'memcached://10.0.0.101',
    'memcached://10.0.0.102',
    // etc...
));
```

The *Data Source Name (DSN)*[9] for this adapter must use the following format:

Listing 13-3
```
memcached://[user:pass@][ip|host|socket[:port]][?weight=int]
```

The DSN must include a IP/host (and an optional port) or a socket path, an optional username and password (for SASL authentication; it requires that the memcached extension was compiled with `--enable-memcached-sasl`) and an optional weight (for prioritizing servers in a cluster; its value is an integer between `0` and `100` which defaults to `null`; a higher value means more priority).

Below are common examples of valid DSNs showing a combination of available values:

Listing 13-4
```php
use Symfony\Component\Cache\Adapter\MemcachedAdapter;

$client = MemcachedAdapter::createConnection(array(
    // hostname + port
    'memcached://my.server.com:11211'

    // hostname without port + SASL username and password
    'memcached://rmf:abcdef@localhost'

    // IP address instead of hostname + weight
    'memcached://127.0.0.1?weight=50'

    // socket instead of hostname/IP + SASL username and password
    'memcached://janesmith:mypassword@/var/run/memcached.sock'

    // socket instead of hostname/IP + weight
    'memcached:///var/run/memcached.sock?weight=20'
));
```

Configure the Options

The *createConnection()*[10] helper method also accepts an array of options as its second argument. The expected format is an associative array of `key => value` pairs representing option names and their respective values:

6. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/MemcachedAdapter.html#method_createConnection
7. http://php.net/manual/en/class.memcached.php
8. https://en.wikipedia.org/wiki/Data_source_name
9. https://en.wikipedia.org/wiki/Data_source_name
10. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/MemcachedAdapter.html#method_createConnection

Listing 13-5

```php
use Symfony\Component\Cache\Adapter\MemcachedAdapter;

$client = MemcachedAdapter::createConnection(
    // a DSN string or an array of DSN strings
    array(),

    // associative array of configuration options
    array(
        'compression' => true,
        'libketama_compatible' => true,
        'serializer' => 'igbinary',
    )
);
```

Available Options

`auto_eject_hosts` **(type:** `bool`**, default:** `false`**)**

Enables or disables a constant, automatic, re-balancing of the cluster by auto-ejecting hosts that have exceeded the configured `server_failure_limit`.

`buffer_writes` **(type:** `bool`**, default:** `false`**)**

Enables or disables buffered input/output operations, causing storage commands to buffer instead of being immediately sent to the remote server(s). Any action that retrieves data, quits the connection, or closes down the connection will cause the buffer to be committed.

`compression` **(type:** `bool`**, default:** `true`**)**

Enables or disables payload compression, where item values longer than 100 bytes are compressed during storage and decompressed during retrieval.

`compression_type` **(type:** `string`**)**

Specifies the compression method used on value payloads. when the **compression** option is enabled.

Valid option values include `fastlz` and `zlib`, with a default value that *varies based on flags used at compilation*.

`connect_timeout` **(type:** `int`**, default:** `1000`**)**

Specifies the timeout (in milliseconds) of socket connection operations when the `no_block` option is enabled.

Valid option values include *any positive integer*.

`distribution` **(type:** `string`**, default:** `consistent`**)**

Specifies the item key distribution method among the servers. Consistent hashing delivers better distribution and allows servers to be added to the cluster with minimal cache losses.

Valid option values include `modula`, `consistent`, and `virtual_bucket`.

`hash` **(type:** `string`**, default:** `md5`**)**

Specifies the hashing algorithm used for item keys. Each hash algorithm has its advantages and its disadvantages. The default is suggested for compatibility with other clients.

Valid option values include `default`, `md5`, `crc`, `fnv1_64`, `fnv1a_64`, `fnv1_32`, `fnv1a_32`, `hsieh`, and `murmur`.

`libketama_compatible` **(type:** `bool`**, default:** `true`**)**
Enables or disables "libketama" compatible behavior, enabling other libketama-based clients to access the keys stored by client instance transparently (like Python and Ruby). Enabling this option sets the `hash` option to `md5` and the `distribution` option to `consistent`.

`no_block` **(type:** `bool`**, default:** `true`**)**
Enables or disables asynchronous input and output operations. This is the fastest transport option available for storage functions.

`number_of_replicas` **(type:** `int`**, default:** `0`**)**

Specifies the number of replicas that should be stored for each item (on different servers). This does not dedicate certain memcached servers to store the replicas in, but instead stores the replicas together with all of the other objects (on the "n" next servers registered).

Valid option values include *any positive integer*.

`prefix_key` **(type:** `string`**, default: an empty string)**

Specifies a "domain" (or "namespace") prepended to your keys. It cannot be longer than 128 characters and reduces the maximum key size.

Valid option values include *any alphanumeric string*.

`poll_timeout` **(type:** `int`**, default:** `1000`**)**

Specifies the amount of time (in seconds) before timing out during a socket polling operation.

Valid option values include *any positive integer*.

`randomize_replica_read` **(type:** `bool`**, type:** `false`**)**
Enables or disables randomization of the replica reads starting point. Normally the read is done from primary server and in case of a miss the read is done from "primary+1", then "primary+2", all the way to "n" replicas. This option sets the replica reads as randomized between all available servers; it allows distributing read load to multiple servers with the expense of more write traffic.

`recv_timeout` **(type:** `int`**, default:** `0`**)**

Specifies the amount of time (in microseconds) before timing out during an outgoing socket (read) operation. When the `no_block` option isn't enabled, this will allow you to still have timeouts on the reading of data.

Valid option values include `0` or *any positive integer*.

`retry_timeout` **(type:** `int`**, default:** `0`**)**

Specifies the amount of time (in seconds) before timing out and retrying a connection attempt.

Valid option values include *any positive integer*.

`send_timeout` **(type:** `int`**, default:** `0`**)**

Specifies the amount of time (in microseconds) before timing out during an incoming socket (send) operation. When the `no_block` option isn't enabled, this will allow you to still have timeouts on the sending of data.

Valid option values include `0` or *any positive integer*.

`serializer` **(type:** `string`**, default:** `php`**)**

Specifies the serializer to use for serializing non-scalar values. The `igbinary` options requires the igbinary PHP extension to be enabled, as well as the memcached extension to have been compiled with support for it.

Valid option values include `php` and `igbinary`.

`server_failure_limit` **(type:** `int`**, default:** `0`**)**

Specifies the failure limit for server connection attempts before marking the server as "dead". The server will remaining in the server pool unless `auto_eject_hosts` is enabled.

Valid option values include *any positive integer*.

`socket_recv_size` **(type:** `int`**)**

Specified the maximum buffer size (in bytes) in the context of incoming (receive) socket connection data.

Valid option values include *any positive integer*, with a default value that *varies by platform and kernel configuration*.

`socket_send_size` **(type:** `int`**)**

Specified the maximum buffer size (in bytes) in the context of outgoing (send) socket connection data.

Valid option values include *any positive integer*, with a default value that *varies by platform and kernel configuration*.

`tcp_keepalive` **(type:** `bool`**, default:** `false`**)**

Enables or disables the "*keep-alive*[11]" *Transmission Control Protocol (TCP)*[12] feature, which is a feature that helps to determine whether the other end has stopped responding by sending probes to the network peer after an idle period and closing or persisting the socket based on the response (or lack thereof).

`tcp_nodelay` **(type:** `bool`**, default:** `false`**)**

Enables or disables the "*no-delay*[13]" (Nagle's algorithm) *Transmission Control Protocol (TCP)*[14] algorithm, which is a mechanism intended to improve the efficiency of networks by reducing the overhead of TCP headers by combining a number of small outgoing messages and sending them all at once.

`use_udp` **(type:** `bool`**, default:** `false`**)**

Enables or disabled the use of *User Datagram Protocol (UDP)*[15] mode (instead of *Transmission Control Protocol (TCP)*[16] mode), where all operations are executed in a "fire-and-forget" manner; no attempt to ensure the operation has been received or acted on will be made once the client has executed it.

Not all library operations are tested in this mode. Mixed TCP and UDP servers are not allowed.

`verify_key` **(type:** `bool`**, default:** `false`**)**

Enables or disables testing and verifying of all keys used to ensure they are valid and fit within the design of the protocol being used.

Reference the *Memcached*[17] extension's *predefined constants*[18] documentation for additional information about the available options.

11. https://en.wikipedia.org/wiki/Keepalive
12. https://en.wikipedia.org/wiki/Transmission_Control_Protocol
13. https://en.wikipedia.org/wiki/TCP_NODELAY
14. https://en.wikipedia.org/wiki/Transmission_Control_Protocol
15. https://en.wikipedia.org/wiki/User_Datagram_Protocol
16. https://en.wikipedia.org/wiki/Transmission_Control_Protocol
17. http://php.net/manual/en/class.memcached.php

18. http://php.net/manual/en/memcached.constants.php

Chapter 14

PDO & Doctrine DBAL Cache Adapter

This adapter stores the cache items in an SQL database. It requires a *PDO*[1], *Doctrine DBAL Connection*[2], or *Data Source Name (DSN)*[3] as its first parameter, and optionally a namespace, default cache lifetime, and options array as its second, third, and forth parameters:

Listing 14-1

```php
use Symfony\Component\Cache\Adapter\PdoAdapter;

$cache = new PdoAdapter(

    // a PDO, a Doctrine DBAL connection or DSN for lazy connecting through PDO
    $databaseConnectionOrDSN,

    // the string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the database table is truncated or its rows are otherwise deleted)
    $defaultLifetime = 0,

    // an array of options for configuring the database connection
    $options = array()
);
```

When passed a *Data Source Name (DSN)*[4] string (instead of a database connection class instance), the connection will be lazy-loaded when needed.

1. http://php.net/manual/en/class.pdo.php
2. https://github.com/doctrine/dbal/blob/master/lib/Doctrine/DBAL/Connection.php
3. https://en.wikipedia.org/wiki/Data_source_name
4. https://en.wikipedia.org/wiki/Data_source_name

Chapter 15
Php Array Cache Adapter

This adapter is a highly performant way to cache static data (e.g. application configuration) that is optimized and preloaded into OPcache memory storage:

Listing 15-1
```php
use Symfony\Component\Cache\Adapter\PhpArrayAdapter;
use Symfony\Component\Cache\Adapter\PhpFilesAdapter;

// somehow, decide it's time to warm up the cache!
if ($needsWarmup) {
    // some static values
    $values = array(
        'stats.num_products' => 4711,
        'stats.num_users' => 1356,
    );

    $cache = new PhpArrayAdapter(
        // single file where values are cached
        __DIR__ . '/somefile.cache',
        // a backup adapter, if you set values after warmup
        new FilesystemAdapter()
    );
    $cache->warmUp($values);
}

// ... then, use the cache!
$cacheItem = $cache->getItem('stats.num_users');
echo $cacheItem->get();
```

This adapter requires PHP 7.x and should be used with the php.ini setting `opcache.enable` on.

Chapter 16
Proxy Cache Adapter

This adapter wraps a *PSR-6*[1] compliant *cache item pool interface*[2]. It is used to integrate your application's cache item pool implementation with the Symfony Cache Component by consuming any implementation of `Psr\Cache\CacheItemPoolInterface`.

This adapter expects a `Psr\Cache\CacheItemPoolInterface` instance as its first parameter, and optionally a namespace and default cache lifetime as its second and third parameters:

Listing 16-1

```php
use Psr\Cache\CacheItemPoolInterface;
use Symfony\Component\Cache\Adapter\ProxyAdapter;

$psr6CachePool = \\ create your own cache pool instance that implements the PSR-6
                 \\ interface `CacheItemPoolInterface`

$cache = new ProxyAdapter(

    // a cache pool instance
    CacheItemPoolInterface $psr6CachePool,

    // a string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until the cache is cleared)
    $defaultLifetime = 0
);
```

1. http://www.php-fig.org/psr/psr-6/
2. http://www.php-fig.org/psr/psr-6/#cacheitempoolinterface

Chapter 17
Redis Cache Adapter

This adapter stores the values in-memory using one (or more) *Redis server*[1] instances. Unlike the APCu adapter, and similarly to the Memcached adapter, it is not limited to the current server's shared memory; you can store contents independent of your PHP environment. The ability to utilize a cluster of servers to provide redundancy and/or fail-over is also available.

Requirements: At least one *Redis server*[2] must be installed and running to use this adapter. Additionally, this adapter requires a compatible extension or library that implements `\Redis`, `\RedisArray`, `RedisCluster`, or `\Predis`.

This adapter expects a *Redis*[3], *RedisArray*[4], *RedisCluster*[5], or *Predis*[6] instance to be passed as the first parameter. A namespace and default cache lifetime can optionally be passed as the second and third parameters:

Listing 17-1
```php
use Symfony\Component\Cache\Adapter\RedisAdapter;

$cache = new RedisAdapter(

    // the object that stores a valid connection to your Redis system
    \Redis $redisConnection,

    // the string prefixed to the keys of the items stored in this cache
    $namespace = '',

    // the default lifetime (in seconds) for cache items that do not define their
    // own lifetime, with a value 0 causing items to be stored indefinitely (i.e.
    // until RedisAdapter::clear() is invoked or the server(s) are purged)
    $defaultLifetime = 0
);
```

1. https://redis.io/
2. https://redis.io/
3. https://github.com/phpredis/phpredis
4. https://github.com/phpredis/phpredis/blob/master/arrays.markdown#readme
5. https://github.com/phpredis/phpredis/blob/master/cluster.markdown#readme
6. https://packagist.org/packages/predis/predis

Configure the Connection

The *`createConnection()`*[7] helper method allows creating and configuring the Redis client class instance using a *Data Source Name (DSN)*[8]:

Listing 17-2
```php
use Symfony\Component\Cache\Adapter\RedisAdapter;

// pass a single DSN string to register a single server with the client
$client = RedisAdapter::createConnection(
    'redis://localhost'
);
```

The DSN can specify either an IP/host (and an optional port) or a socket path, as well as a user and password and a database index.

A *Data Source Name (DSN)*[9] for this adapter must use the following format.

Listing 17-3
```
redis://[user:pass@][ip|host|socket[:port]][/db-index]
```

Below are common examples of valid DSNs showing a combination of available values:

Listing 17-4
```php
use Symfony\Component\Cache\Adapter\RedisAdapter;

// host "my.server.com" and port "6379"
RedisAdapter::createConnection('redis://my.server.com:6379');

// host "my.server.com" and port "6379" and database index "20"
RedisAdapter::createConnection('redis://my.server.com:6379/20');

// host "localhost" and SASL use "rmf" and pass "abcdef"
RedisAdapter::createConnection('redis://rmf:abcdef@localhost');

// socket "/var/run/redis.sock" and SASL user "user1" and pass "bad-pass"
RedisAdapter::createConnection('redis://user1:bad-pass@/var/run/redis.sock');
```

Configure the Options

The *`createConnection()`*[10] helper method also accepts an array of options as its second argument. The expected format is an associative array of `key => value` pairs representing option names and their respective values:

Listing 17-5
```php
use Symfony\Component\Cache\Adapter\RedisAdapter;

$client = RedisAdapter::createConnection(

    // provide a string dsn
    'redis://localhost:6739',

    // associative array of configuration options
    array(
        'persistent' => 0,
        'persistent_id' => null,
        'timeout' => 30,
        'read_timeout' => 0,
```

7. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/RedisAdapter.html#method_createConnection
8. https://en.wikipedia.org/wiki/Data_source_name
9. https://en.wikipedia.org/wiki/Data_source_name
10. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/RedisAdapter.html#method_createConnection

```
14          'retry_interval' => 0,
15      )
16
17  );
```

Available Options

`class` (type: `string`)
: Specifies the connection library to return, either `\Redis` or `\Predis\Client`. If none is specified, it will return `\Redis` if the `redis` extension is available, and `\Predis\Client` otherwise.

`persistent` (type: `int`, default: `0`)
: Enables or disables use of persistent connections. A value of `0` disables persistent connections, and a value of `1` enables them.

`persistent_id` (type: `string|null`, default: `null`)
: Specifies the persistent id string to use for a persistent connection.

`read_timeout` (type: `int`, default: `0`)
: Specifies the time (in seconds) used when performing read operations on the underlying network resource before the operation times out.

`retry_interval` (type: `int`, default: `0`)
: Specifies the delay (in milliseconds) between reconnection attempts in case the client loses connection with the server.

`timeout` (type: `int`, default: `30`)
: Specifies the time (in seconds) used to connect to a Redis server before the connection attempt times out.

When using the *Predis*[11] library some additional Predis-specific options are available. Reference the *Predis Connection Parameters*[12] documentation for more information.

11. https://packagist.org/packages/predis/predis
12. https://github.com/nrk/predis/wiki/Connection-Parameters#list-of-connection-parameters

Chapter 18

Adapters For Interoperability between PSR-6 and PSR-16 Cache

Sometimes, you may have a Cache object that implements the PSR-16 standard, but need to pass it to an object that expects a PSR-6 cache adapter. Or, you might have the opposite situation. The cache component contains two classes for bidirectional interoperability between PSR-6 and PSR-16 caches.

Using a PSR-16 Cache Object as a PSR-6 Cache

Suppose you want to work with a class that requires a PSR-6 Cache pool object. For example:

Listing 18-1

```php
use Psr\Cache\CacheItemPoolInterface;

// just a made-up class for the example
class GitHubApiClient
{
    // ...

    // this requires a PSR-6 cache object
    public function __construct(CacheItemPoolInterface $cachePool)
    {
        // ...
    }
}
```

But, you already have a PSR-16 cache object, and you'd like to pass this to the class instead. No problem! The Cache component provides the *`SimpleCacheAdapter`*[1] class for exactly this use-case:

Listing 18-2

```php
use Symfony\Component\Cache\Simple\FilesystemCache;
use Symfony\Component\Cache\Adapter\SimpleCacheAdapter

// the PSR-16 cache object that you want to use
$psr16Cache = new FilesystemCache();

// a PSR-6 cache that uses your cache internally!
```

1. http://api.symfony.com/4.0/Symfony/Component/Cache/Adapter/SimpleCacheAdapter.html

```php
$psr6Cache = new SimpleCacheAdapter($psr16Cache);

// now use this wherever you want
$githubApiClient = new GitHubApiClient($psr6Cache);
```

Using a PSR-6 Cache Object as a PSR-16 Cache

Suppose you want to work with a class that requires a PSR-16 Cache object. For example:

Listing 18-3

```php
use Psr\SimpleCache\CacheInterface;

// just a made-up class for the example
class GitHubApiClient
{
    // ...

    // this requires a PSR-16 cache object
    public function __construct(CacheInterface $cache)
    {
        // ...
    }
}
```

But, you already have a PSR-6 cache pool object, and you'd like to pass this to the class instead. No problem! The Cache component provides the *Psr6Cache*[2] class for exactly this use-case:

Listing 18-4

```php
use Symfony\Component\Cache\Adapter\FilesystemAdapter;
use Symfony\Component\Cache\Simple\Psr6Cache;

// the PSR-6 cache object that you want to use
$psr6Cache = new FilesystemAdapter();

// a PSR-16 cache that uses your cache internally!
$psr16Cache = new Psr6Cache($psr6Cache);

// now use this wherever you want
$githubApiClient = new GitHubApiClient($psr16Cache);
```

2. http://api.symfony.com/4.0/Symfony/Component/Cache/Simple/Psr6Cache.html

Chapter 19
The ClassLoader Component

The ClassLoader component was removed in Symfony 4.0. As an alternative, use any of the *class loading optimizations*[1] provided by Composer.

1. https://getcomposer.org/doc/articles/autoloader-optimization.md

Chapter 20
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 21
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 21-1

```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.yaml' file paths
    $yamlUserFiles = ...;

    $resources = array();

    foreach ($yamlUserFiles as $yamlUserFile) {
        // see the article "Loading resources" to
        // know 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/4.0/Symfony/Component/Config/ConfigCache.html

```
}

// you may want to require the cached code:
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 22

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 22-1

```yaml
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 22-2
```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 22-3
```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/4.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
2. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
3. http://api.symfony.com/4.0/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 22-4
```
$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 22-5
```
$rootNode
    ->children()
        ->enumNode('delivery')
            ->values(array('standard', 'expedited', 'priority'))
        ->end()
    ->end()
;
```

This will restrict the `delivery` options to be either `standard`, `expedited` or `priority`.

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 22-6
```
$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 22-7
```
$rootNode
    ->children()
        ->arrayNode('connections')
            ->arrayPrototype()
```

4. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/IntegerNodeDefinition.html#method_min
5. http://api.symfony.com/4.0/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.).

Sometimes, to improve the user experience of your application or bundle, you may allow to use a simple string or numeric value where an array value is required. Use the `castToArray()` helper to turn those variables into arrays:

Listing 22-8
```
->arrayNode('hosts')
    ->beforeNormalization()->castToArray()->end()
    // ...
->end()
```

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.

`ignoreExtraKeys()`
: Allows extra config keys to be specified under an array without throwing an exception.

A basic prototyped array configuration can be defined as follows:

Listing 22-9
```
$node
    ->fixXmlConfig('driver')
    ->children()
        ->arrayNode('drivers')
            ->scalarPrototype()->end()
        ->end()
    ->end()
;
```

When using the following YAML configuration:

Listing 22-10
```
drivers: ['mysql', 'sqlite']
```

Or the following XML configuration:

Listing 22-11
```xml
<driver>mysql</driver>
<driver>sqlite</driver>
```

The processed configuration is:

Listing 22-12
```
Array(
    [0] => 'mysql'
    [1] => 'sqlite'
)
```

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

Listing 22-13
```
$node
    ->fixXmlConfig('connection')
    ->children()
        ->arrayNode('connections')
            ->arrayPrototype()
                ->children()
                    ->scalarNode('table')->end()
                    ->scalarNode('user')->end()
                    ->scalarNode('password')->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

When using the following YAML configuration:

Listing 22-14
```yaml
connections:
    - { table: symfony, user: root, password: ~ }
    - { table: foo, user: root, password: pa$$ }
```

Or the following XML configuration:

Listing 22-15
```xml
<connection table="symfony" user="root" password="null" />
<connection table="foo" user="root" password="pa$$" />
```

The processed configuration is:

Listing 22-16
```
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 22-17
```yaml
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 22-18
```php
$node
    ->fixXmlConfig('connection')
    ->children()
        ->arrayNode('connections')
            ->useAttributeAsKey('name')
            ->arrayPrototype()
                ->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 22-19
```xml
<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 22-20
```
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 22-21
```
$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()
;
```

Deprecating the Option

You can deprecate options using the *`setDeprecated()`*[6] method:

Listing 22-22
```
$rootNode
    ->children()
        ->integerNode('old_option')
            // this outputs the following generic deprecation message:
            // The child node "old_option" at path "..." is deprecated.
            ->setDeprecated()

            // you can also pass a custom deprecation message (%node% and %path% placeholders are available):
            ->setDeprecated('The "%node%" option is deprecated. Use "new_config_option" instead.')
        ->end()
    ->end()
;
```

If you use the Web Debug Toolbar, these deprecation notices are shown when the configuration is rebuilt.

6. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/NodeDefinition.html#method_setDeprecated

Documenting the Option

All options can be documented using the *info()*[7] method.

Listing 22-23
```php
$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 22-24
```yaml
# This value is only used for the search results page.
entries_per_page:     25
```

and in XML:

Listing 22-25
```xml
<!-- entries-per-page: This value is only used for the search results page. -->
<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()*[8] and *canBeDisabled()*[9] methods:

Listing 22-26
```php
$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:

7. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/NodeDefinition.html#method_info
8. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#method_canBeEnabled
9. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#method_canBeDisabled

`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 22-27
```php
public function getConfigTreeBuilder()
{
    $treeBuilder = new TreeBuilder();
    $rootNode = $treeBuilder->root('database');

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

    return $treeBuilder;
}

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

    $node
        ->isRequired()
        ->requiresAtLeastOneElement()
        ->useAttributeAsKey('name')
        ->arrayPrototype()
            ->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 22-28
```
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 22-29
```
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 22-30
```
<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 22-31
```
$rootNode
    ->fixXmlConfig('extension')
    ->children()
        ->arrayNode('extensions')
            ->scalarPrototype()->end()
        ->end()
    ->end()
;
```

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

Listing 22-32
```
$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 22-33
```
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 22-34
```
<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 22-35
```
connection:
    name:     my_mysql_connection
    host:     localhost
    driver:   mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 22-36
```
connection: my_mysql_connection
```

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

Listing 22-37
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
                ->ifString()
                ->then(function ($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`*[10]. 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 22-38

10. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/ExprBuilder.html

```
$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()`
- `ifEmpty()` (since Symfony 3.2)
- `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`*[11] uses the tree as it was built using the *`TreeBuilder`*[12] 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 22-39
```
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.yaml')
);
$config2 = Yaml::parse(
    file_get_contents(__DIR__.'/src/Matthias/config/config_extra.yaml')
);

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

$processor = new Processor();
```

11. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Processor.html
12. http://api.symfony.com/4.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html

```
15 $configuration = new DatabaseConfiguration();
16 $processedConfiguration = $processor->processConfiguration(
17     $configuration,
18     $configs
19 );
```

Chapter 23
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 23-1
```php
use Symfony\Component\Config\FileLocator;

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

$locator = new FileLocator($configDirectories);
$yamlUserFiles = $locator->locate('users.yaml', 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/4.0/Symfony/Component/Config/FileLocator.html
3. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/LoaderInterface.html
4. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/FileLoader.html

Listing 23-2
```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.yaml');
    }

    public function supports($resource, $type = null)
    {
        return is_string($resource) && 'yaml' === 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 23-3
```php
use Symfony\Component\Config\Loader\LoaderResolver;
use Symfony\Component\Config\Loader\DelegatingLoader;

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

// YamlUserLoader is used to load this resource because it supports
// files with the '.yaml' extension
$delegatingLoader->load(__DIR__.'/users.yaml');
```

5. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/LoaderResolver.html
6. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/DelegatingLoader.html
7. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/LoaderResolver.html
8. http://api.symfony.com/4.0/Symfony/Component/Config/Loader/LoaderResolver.html