Symfony

The Components Book

Version: master

generated on May 21, 2019

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! Your command line in that case is `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, 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

Listing 2-2
```
$ composer require symfony/asset
```

If you install this component outside of a Symfony application, you must require the `vendor/autoload.php` file in your code to enable the class autoloading mechanism provided by Composer. Read *this article* for more details.

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 2-3

```
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*[1], which defines the following two methods:

getVersion()[2]
: Returns the asset version for an asset.

getUrl()[3]
: 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*[4], which doesn't add any version to the asset and *StaticVersionStrategy*[5], 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 2-4

```
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\VersionStrategy\StaticVersionStrategy;
```

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/PackageInterface.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/PackageInterface.php
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/PackageInterface.php
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/VersionStrategy/EmptyVersionStrategy.php
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/VersionStrategy/StaticVersionStrategy.php

```
3
4  $package = new Package(new StaticVersionStrategy('v1'));
5
6  // Absolute path
7  echo $package->getUrl('/image.png');
8  // result: /image.png?v1
9
10 // Relative path
11 echo $package->getUrl('image.png');
12 // 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 2-5

```
1  // puts the 'version' word before the version value
2  $package = new Package(new StaticVersionStrategy('v1', '%s?version=%s'));
3
4  echo $package->getUrl('/image.png');
5  // result: /image.png?version=v1
6
7  // puts the asset version before its path
8  $package = new Package(new StaticVersionStrategy('v1', '%2$s/%1$s'));
9
10 echo $package->getUrl('/image.png');
11 // result: /v1/image.png
12
13 echo $package->getUrl('image.png');
14 // result: v1/image.png
```

JSON File Manifest

A popular strategy to manage asset versioning, which is used by tools such as *Webpack*[6], is to generate a JSON file mapping all source file names to their corresponding output file:

Listing 2-6

```
1  // rev-manifest.json
2  {
3      "css/app.css": "build/css/app.b916426ea1d10021f3f17ce8031f93c2.css",
4      "js/app.js": "build/js/app.13630905267b809161e71d0f8a0c017b.js",
5      "...": "..."
6  }
```

In those cases, use the *`JsonManifestVersionStrategy`*[7]:

Listing 2-7

```
1  use Symfony\Component\Asset\Package;
2  use Symfony\Component\Asset\VersionStrategy\JsonManifestVersionStrategy;
3
4  $package = new Package(new JsonManifestVersionStrategy(__DIR__.'/rev-manifest.json'));
5
6  echo $package->getUrl('css/app.css');
7  // result: build/css/app.b916426ea1d10021f3f17ce8031f93c2.css
```

Custom Version Strategies

Use the *`VersionStrategyInterface`*[8] 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 2-8

```
1  use Symfony\Component\Asset\VersionStrategy\VersionStrategyInterface;
2
```

6. https://webpack.js.org/
7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/VersionStrategy/JsonManifestVersionStrategy.php
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/VersionStrategy/VersionStrategyInterface.php

```
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*[9] class with *PathPackage*[10] to avoid repeating that path over and over again:

Listing 2-9

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

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

echo $pathPackage->getUrl('logo.png');
// result: /static/images/logo.png?v1

// Base path is ignored when using absolute paths
echo $pathPackage->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 2-10

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

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

echo $pathPackage->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 $pathPackage->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

9. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/Package.php
10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/PathPackage.php

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`*[11] class to generate absolute URLs for their assets:

Listing 2-11
```php
use Symfony\Component\Asset\UrlPackage;
// ...

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

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

You can also pass a schema-agnostic URL:

Listing 2-12
```php
use Symfony\Component\Asset\UrlPackage;
// ...

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

echo $urlPackage->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. If you want to use this, 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 2-13
```php
use Symfony\Component\Asset\UrlPackage;
// ...

$urls = [
    '//static1.example.com/images/',
    '//static2.example.com/images/',
];
$urlPackage = new UrlPackage($urls, new StaticVersionStrategy('v1'));

echo $urlPackage->getUrl('/logo.png');
// result: http://static1.example.com/images/logo.png?v1
echo $urlPackage->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):

11. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/UrlPackage.php

Listing 2-14
```php
use Symfony\Component\Asset\Context\RequestStackContext;
use Symfony\Component\Asset\UrlPackage;
// ...

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

echo $urlPackage->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*[12] class to simplify management of several packages.

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

Listing 2-15
```php
use Symfony\Component\Asset\Package;
use Symfony\Component\Asset\Packages;
use Symfony\Component\Asset\PathPackage;
use Symfony\Component\Asset\UrlPackage;
// ...

$versionStrategy = new StaticVersionStrategy('v1');

$defaultPackage = new Package($versionStrategy);

$namedPackages = [
    '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 2-16
```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
```

Local Files and Other Protocols

In addition to HTTP this component supports other protocols (such as `file://` and `ftp://`). This allows for example to serve local files in order to improve performance:

Listing 2-17

12. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Asset/Packages.php

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

$localPackage = new UrlPackage(
    'file:///path/to/images/',
    new EmptyVersionStrategy()
);

$ftpPackage = new UrlPackage(
    'ftp://example.com/images/',
    new EmptyVersionStrategy()
);

echo $localPackage->getUrl('/logo.png');
// result: file:///path/to/images/logo.png

echo $ftpPackage->getUrl('/logo.png');
// result: ftp://example.com/images/logo.png
```

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

Listing 3-1
```
1 $ composer require symfony/browser-kit
```

If you install this component outside of a Symfony application, you must require the `vendor/autoload.php` file in your code to enable the class autoloading mechanism provided by Composer. Read *this article* for more details.

Basic Usage

This article explains how to use the BrowserKit features as an independent component in any PHP application. Read the Symfony Functional Tests article to learn about how to use it in Symfony applications.

1. https://github.com/FriendsOfPHP/Goutte

Creating a Client

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

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

Listing 3-2
```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*[3]. For an implementation based on `HttpKernelInterface`, have a look at the *`Client`*[4] provided by the *HttpKernel component*.

Making Requests

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

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

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

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

The *`xmlHttpRequest()`*[7] method, which defines the same arguments as the `request()` method, is a shortcut to make AJAX requests:

Listing 3-4
```php
use Acme\Client;

$client = new Client();
// the required HTTP_X_REQUESTED_WITH header is added automatically
$crawler = $client->xmlHttpRequest('GET', '/');
```

Clicking Links

The `Client` object is capable of simulating link clicks. Pass the text content of the link and the client will perform the needed HTTP GET request to simulate the link click:

---

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/BrowserKit/Client.php
3. https://github.com/FriendsOfPHP/Goutte
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpKernel/Client.php
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/BrowserKit/Client.php
6. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/DomCrawler/Crawler.php
7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/BrowserKit/Client.php

Listing 3-5
```php
use Acme\Client;

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

$crawler = $client->clickLink('Go elsewhere...');
```

If you need the *Link*[8] object that provides access to the link properties (e.g. `$link->getMethod()`, `$link->getUri()`), use this other method:

```
// ... $crawler = $client->request('GET', '/product/123'); $link = $crawler->selectLink('Go elsewhere...')->link(); $client->click($link);
```

Submitting Forms

The `Client` object is also capable of submitting forms. First, select the form using any of its buttons and then override any of its properties (method, field values, etc.) before submitting it:

Listing 3-6
```php
use Acme\Client;

$client = new Client();
$crawler = $client->request('GET', 'https://github.com/login');

// find the form with the 'Log in' button and submit it
// 'Log in' can be the text content, id, value or name of a <button> or <input type="submit">
$client->submitForm('Log in');

// the second optional argument lets you override the default form field values
$client->submitForm('Log in', [
    'login' => 'my_user',
    'password' => 'my_pass',
    // to upload a file, the value must be the absolute file path
    'file' => __FILE__,
]);

// you can override other form options too
$client->submitForm(
    'Log in',
    ['login' => 'my_user', 'password' => 'my_pass'],
    // override the default form HTTP method
    'PUT',
    // override some $_SERVER parameters (e.g. HTTP headers)
    ['HTTP_ACCEPT_LANGUAGE' => 'es']
);
```

If you need the *Form*[9] object that provides access to the form properties (e.g. `$form->getUri()`, `$form->getValues()`, `$form->getFields()`), use this other method:

Listing 3-7
```php
// ...

// 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);
```

8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/DomCrawler/Link.php
9. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/DomCrawler/Form.php

Cookies

Retrieving Cookies

The `Client` implementation exposes cookies (if any) through a *`CookieJar`*[10], which allows you to store and retrieve any cookie while making requests with the client:

Listing 3-8

```
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();
$rawValue   = $cookie->getRawValue();
$isSecure   = $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 3-9

```
use Acme\Client;

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

// 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) {
```

10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/BrowserKit/CookieJar.php

```
25      // ...
26  }
```

Setting Cookies

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

Listing 3-10
```
1  use Acme\Client;
2
3  // create cookies and add to cookie jar
4  $cookie = new Cookie('flavor', 'chocolate', strtotime('+1 day'));
5  $cookieJar = new CookieJar();
6  $cookieJar->set($cookie);
7
8  // create a client and set the cookies
9  $client = new Client([], null, $cookieJar);
10 // ...
```

History

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

Listing 3-11
```
1  use Acme\Client;
2
3  $client = new Client();
4  $client->request('GET', '/');
5
6  // select and click on a link
7  $link = $crawler->selectLink('Documentation')->link();
8  $client->click($link);
9
10 // go back to home page
11 $crawler = $client->back();
12
13 // go forward to documentation page
14 $crawler = $client->forward();
```

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

Listing 3-12
```
1  use Acme\Client;
2
3  $client = new Client();
4  $client->request('GET', '/');
5
6  // reset the client (history and cookies are cleared too)
7  $client->restart();
```

Learn more

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

Chapter 4
The Cache Component

> The Cache component provides features covering simple to advanced caching needs. It natively implements *PSR-6*[1] and the *Cache Contracts*[2] for greatest interoperability. It is designed for performance and resiliency, ships with ready to use adapters for the most common caching backends. It enables tag-based invalidation and cache stampede protection via locking and early expiration.

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

Installation

Listing 4-1
```
$ composer require symfony/cache
```

If you install this component outside of a Symfony application, you must require the `vendor/autoload.php` file in your code to enable the class autoloading mechanism provided by Composer. Read *this article* for more details.

Cache Contracts versus PSR-6

This component includes *two* different approaches to caching:

PSR-6 Caching:

A generic cache system, which involves cache "pools" and cache "items".

1. http://www.php-fig.org/psr/psr-6/
2. https://github.com/symfony/contracts/blob/master/Cache/CacheInterface.php

Cache Contracts:
A simpler yet more powerful way to cache values based on recomputation callbacks.

Using the Cache Contracts approach is recommended: it requires less code boilerplate and provides cache stampede protection by default.

Cache Contracts

All adapters support the Cache Contracts. They contain only two methods: `get()` and `delete()`. There's no `set()` method because the `get()` method both gets and sets the cache values.

The first thing you need is to instantiate a cache adapter. The *`FilesystemAdapter`*[3] is used in this example:

Listing 4-2
```php
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter();
```

Now you can retrieve and delete cached data using this object. The first argument of the `get()` method is a key, an arbitrary string that you associate to the cached value so you can retrieve it later. The second argument is a PHP callable which is executed when the key is not found in the cache to generate and return the value:

Listing 4-3
```php
use Symfony\Contracts\Cache\ItemInterface;

// The callable will only be executed on a cache miss.
$value = $cache->get('my_cache_key', function (ItemInterface $item) {
    $item->expiresAfter(3600);

    // ... do some HTTP request or heavy computations
    $computedValue = 'foobar';

    return $computedValue;
});

echo $value; // 'foobar'

// ... and to remove the cache key
$cache->delete('my_cache_key');
```

Use cache tags to delete more than one key at the time. Read more at *Cache Invalidation*.

The Cache Contracts also comes with built in *Stampede prevention*[4]. This will remove CPU spikes at the moments when the cache is cold. If an example application spends 5 seconds to compute data that is cached for 1 hour and this data is accessed 10 times every second, this means that you mostly have cache hits and everything is fine. But after 1 hour, we get 10 new requests to a cold cache. So the data is computed again. The next second the same thing happens. So the data is computed about 50 times before the cache is warm again. This is where you need stampede prevention

The first solution is to use locking: only allow one PHP process (on a per-host basis) to compute a specific key at a time. Locking is built-in by default, so you don't need to do anything beyond leveraging the Cache Contracts.

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/FilesystemAdapter.php
4. https://en.wikipedia.org/wiki/Cache_stampede

The second solution is also built-in when using the Cache Contracts: instead of waiting for the full delay before expiring a value, recompute it ahead of its expiration date. The *Probabilistic early expiration*[5] algorithm randomly fakes a cache miss for one user while others are still served the cached value. You can control its behavior with the third optional parameter of `get()`[6], which is a float value called "beta".

By default the beta is `1.0` and higher values mean earlier recompute. Set it to `0` to disable early recompute and set it to `INF` to force an immediate recompute:

Listing 4-4
```php
use Symfony\Contracts\Cache\ItemInterface;

$beta = 1.0;
$value = $cache->get('my_cache_key', function (ItemInterface $item) {
    $item->expiresAfter(3600);
    $item->tag(['tag_0', 'tag_1']);

    return '...';
}, $beta);
```

Available Cache Adapters

The following cache adapters are available:

- 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
- Php Files Cache Adapter
- Proxy Cache Adapter
- Redis Cache Adapter

Generic Caching (PSR-6)

To use the generic 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; see the *Cache Items* article for more details.

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.)

5. https://en.wikipedia.org/wiki/Cache_stampede#Probabilistic_early_expiration
6. https://github.com/symfony/symfony/blob/master/src/Symfony/Contracts/Cache/CacheInterface.php

Basic Usage (PSR-6)

This part of the component is an implementation of *PSR-6*[7], which means that its basic API is the same as defined in the document. 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`*[8]:

Listing 4-5
```
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter();
```

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

Listing 4-6
```
// create a new item by trying to get it from the cache
$productsCount = $cache->getItem('stats.products_count');

// assign a value to the item and save it
$productsCount->set(4711);
$cache->save($productsCount);

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

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

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

Advanced Usage

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

7. http://www.php-fig.org/psr/psr-6/
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/FilesystemAdapter.php

Chapter 5
APCu Cache Adapter

This adapter is a high-performance, shared memory cache. It can *significantly* increase an application's performance, as its cache contents are stored in shared memory, a component appreciably faster than many others, such as the filesystem.

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

The ApcuAdapter can optionally be provided a namespace, default cache lifetime, and cache items version string as constructor arguments:

Listing 5-1
```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
);
```

Use of this adapter is discouraged in write/delete heavy workloads, as these operations cause memory fragmentation that results in significantly degraded performance.

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

This adapter's CRUD operations are specific to the PHP SAPI it is running under. This means cache operations (such as additions, deletions, etc) using the CLI will not be available under the FPM or CGI SAPIs.

Chapter 6
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 6-1

```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. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/ArrayAdapter.php

Chapter 7
Chain Cache Adapter

This adapter allows combining any number of the other available cache adapters. Cache items are fetched from the first adapter containing them and cache items are saved to all the given adapters. This exposes a simple and efficient method for creating a layered cache.

The ChainAdapter must be provided an array of adapters and optionally a maximum cache lifetime as its constructor arguments:

Listing 7-1

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

$cache = new ChainAdapter([

    // 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 to 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 7-2

```php
use Symfony\Component\Cache\Adapter\ApcuAdapter;
use Symfony\Component\Cache\Adapter\ChainAdapter;
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

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

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/ApcuAdapter.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/FilesystemAdapter.php

When calling this adapter's *prune()*[3] method, the call is delegated to all its compatible cache adapters. It is safe to mix both adapters that *do* and do *not* implement *PruneableInterface*[4], as incompatible adapters are silently ignored:

Listing 7-3

```php
use Symfony\Component\Cache\Adapter\ApcuAdapter;
use Symfony\Component\Cache\Adapter\ChainAdapter;
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new ChainAdapter([
    new ApcuAdapter(),        // does NOT implement PruneableInterface
    new FilesystemAdapter(),  // DOES implement PruneableInterface
]);

// prune will proxy the call to FilesystemAdapter while silently skipping ApcuAdapter
$cache->prune();
```

Since Symfony 3.4, this adapter implements *PruneableInterface*[5], allowing for manual pruning of expired cache entries by calling its `prune()` method.

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/ChainAdapter.php
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/PruneableInterface.php
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/PruneableInterface.php

Chapter 8
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 8-1

```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');

$cache = 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
);
```

A *`DoctrineProvider`*[2] class is also provided by the component to use any PSR6-compatible implementations with Doctrine-compatible classes.

1. https://github.com/doctrine/cache
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/DoctrineProvider.php

Chapter 9
Filesystem Cache Adapter

This adapter offers improved application performance for those who cannot install tools like APCu or Redis in their environment. It stores the cache item expiration and content as regular files in a collection of directories on a locally mounted filesystem.

The performance of this adapter can be greatly increased by utilizing a temporary, in-memory filesystem, such as *tmpfs*[1] on Linux, or one of the many other *RAM disk solutions*[2] available.

The FilesystemAdapter can optionally be provided a namespace, default cache lifetime, and cache root path as constructor parameters:

Listing 9-1
```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
);
```

The overhead of filesystem IO often makes this adapter one of the *slower* choices. If throughput is paramount, the in-memory adapters (Apcu, Memcached, and Redis) or the database adapters (Doctrine and PDO) are recommended.

1. https://wiki.archlinux.org/index.php/tmpfs
2. https://en.wikipedia.org/wiki/List_of_RAM_drive_software

Since Symfony 3.4, this adapter implements *`PruneableInterface`*[3], enabling manual pruning of expired cache items by calling its `prune()` method.

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/PruneableInterface.php

Chapter 10
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 10-1

```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 10-2
```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'
    // the DSN can include config options (pass them as a query string):
    // 'memcached://localhost:11222?retry_timeout=10'
    // 'memcached://localhost:11222?socket_recv_size=1&socket_send_size=2'
);

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

// a single DSN can define multiple servers using the following syntax:
// host[hostname-or-IP:port] (where port is optional). Sockets must include a trailing ':'
$client = MemcachedAdapter::createConnection(
    'memcached:?host[localhost]&host[localhost:12345]&host[/some/memcached.sock:]=3'
);
```

New in version 4.2: The option to define multiple servers in a single DSN was introduced in Symfony 4.2.

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

Listing 10-3
```
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 10-4
```php
use Symfony\Component\Cache\Adapter\MemcachedAdapter;

$client = MemcachedAdapter::createConnection([
    // 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'
]);
```

6. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/MemcachedAdapter.php
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

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 10-5

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

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

    // associative array of configuration options
    [
        '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`.

10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/MemcachedAdapter.php

`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 disables 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.

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

`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.

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

Chapter 11
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 11-1
```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 table and connection
    $options = []
);
```

The table where values are stored is created automatically on the first call to the `save()`[4] method. You can also create this table explicitly by calling the `createTable()`[5] method in your code.

When passed a *Data Source Name (DSN)*[6] 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://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/PdoAdapter.php
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/PdoAdapter.php
6. https://en.wikipedia.org/wiki/Data_source_name

Since Symfony 3.4, this adapter implements *`PruneableInterface`*[7], allowing for manual pruning of expired cache entries by calling its `prune()` method.

7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/PruneableInterface.php

Chapter 12
Php Array Cache Adapter

This adapter is a high performance cache for static data (e.g. application configuration) that is optimized and preloaded into OPcache memory storage. It is suited for any data that is mostly read-only after warmup:

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

// somehow, decide it's time to warm up the cache!
if ($needsWarmup) {
    // some static values
    $values = [
        'stats.products_count' => 4711,
        'stats.users_count' => 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.users_count');
echo $cacheItem->get();
```

This adapter requires turning on the `opcache.enable` php.ini setting.

Chapter 13
Php Files Cache Adapter

Similarly to Filesystem Adapter, this cache implementation writes cache entries out to disk, but unlike the Filesystem cache adapter, the PHP Files cache adapter writes and reads back these cache files *as native PHP code*. For example, caching the value `['my', 'cached', 'array']` will write out a cache file similar to the following:

Listing 13-1
```php
<?php return [

    // the cache item expiration
    0 => 9223372036854775807,

    // the cache item contents
    1 => [
        0 => 'my',
        1 => 'cached',
        2 => 'array',
    ],

];
```

This adapter requires turning on the `opcache.enable` php.ini setting. As cache items are included and parsed as native PHP code and due to the way *OPcache*[1] handles file includes, this adapter has the potential to be much faster than other filesystem-based caches.

While it supports updates and because it is using OPcache as a backend, this adapter is better suited for append-mostly needs. Using it in other scenarios might lead to periodical reset of the OPcache memory, potentially leading to degraded performance.

The PhpFilesAdapter can optionally be provided a namespace, default cache lifetime, and cache directory path as constructor arguments:

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

$cache = new PhpFilesAdapter(

```

1. http://php.net/manual/en/book.opcache.php

```
    // 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
);
```

Since Symfony 3.4, this adapter implements *`PruneableInterface`*[2], allowing for manual pruning of expired cache entries by calling its `prune()` method.

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/PruneableInterface.php

Chapter 14
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`.

It can also be used to prefix all keys automatically before storing items in the decorated pool, effectively allowing the creation of several namespaced pools out of a single one.

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 14-1
```php
use Psr\Cache\CacheItemPoolInterface;
use Symfony\Component\Cache\Adapter\ProxyAdapter;

// create your own cache pool instance that implements
// the PSR-6 CacheItemPoolInterface
$psr6CachePool = ...

$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 15
Redis Cache Adapter

This article explains how to configure the Redis adapter when using the Cache as an independent component in any PHP application. Read the Symfony Cache configuration article if you are using it in a Symfony application.

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 15-1
```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.
```

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

```
    // until RedisAdapter::clear() is invoked or the server(s) are purged)
    $defaultLifetime = 0
);
```

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 15-2
```
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 password and a database index.

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

Listing 15-3
```
redis://[pass@][ip|host|socket[:port]][/db-index]
```

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

Listing 15-4
```
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", auth "abcdef" and timeout 5 seconds
RedisAdapter::createConnection('redis://abcdef@localhost?timeout=5');

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

// a single DSN can define multiple servers using the following syntax:
// host[hostname-or-IP:port] (where port is optional). Sockets must include a trailing ':'
RedisAdapter::createConnection(
    'redis:?host[localhost]&host[localhost:6379]&host[/var/run/redis.sock:]&auth=my-password&redis_cluster=1'
);
```

New in version 4.2: The option to define multiple servers in a single DSN was introduced in Symfony 4.2.

See the *`RedisTrait`*[10] for more options you can pass as DSN parameters.

7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/RedisAdapter.php
8. https://en.wikipedia.org/wiki/Data_source_name
9. https://en.wikipedia.org/wiki/Data_source_name
10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Traits/RedisTrait.php

Configure the Options

The *`createConnection()`*[11] 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 15-5

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

$client = RedisAdapter::createConnection(

    // provide a string dsn
    'redis://localhost:6379',

    // associative array of configuration options
    [
        'compression' => true,
        'lazy' => false,
        'persistent' => 0,
        'persistent_id' => null,
        'tcp_keepalive' => 0,
        'timeout' => 30,
        'read_timeout' => 0,
        'retry_interval' => 0,
    ]

);
```

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.

`compression` (type: `bool`, default: `true`)
: Enables or disables compression of items. This requires phpredis v4 or higher with LZF support enabled.

`lazy` (type: `bool`, default: `false`)
: Enables or disables lazy connections to the backend. It's `false` by default when using this as a standalone component and `true` by default when using it inside a Symfony application.

`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.

11. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/RedisAdapter.php

`tcp_keepalive` **(type: `int`, default: `0`)**
: Specifies the *TCP-keepalive*[12] timeout (in seconds) of the connection. This requires phpredis v4 or higher and a TCP-keepalive enabled 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*[13] library some additional Predis-specific options are available. Reference the *Predis Connection Parameters*[14] documentation for more information.

12. https://redis.io/topics/clients#tcp-keepalive
13. https://packagist.org/packages/predis/predis
14. https://github.com/nrk/predis/wiki/Connection-Parameters#list-of-connection-parameters

Chapter 16

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

Sometimes, you may have a Cache object that implements the *PSR-16*[1] 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 16-1
```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 *`Psr16Adapter`*[2] class for exactly this use-case:

Listing 16-2
```php
use Symfony\Component\Cache\Adapter\Psr16Adapter;

// $psr16Cache is the PSR-16 object that you want to use as a PSR-6 one

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

1. http://www.php-fig.org/psr/psr-16/
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Adapter/Psr16Adapter.php

```php
$psr6Cache = new Psr16Adapter($psr16Cache);

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

New in version 4.3: The `Psr16Adapter` class was introduced in Symfony 4.3.

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 16-3
```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 *`Psr16Cache`*[3] class for exactly this use-case:

Listing 16-4
```php
use Symfony\Component\Cache\Adapter\FilesystemAdapter;
use Symfony\Component\Cache\Psr16Cache;

// 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);
```

New in version 4.3: The `Psr16Cache` class was introduced in Symfony 4.3.

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Cache/Psr16Cache.php

Chapter 17
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 18
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

Listing 18-1
```
1 $ composer require symfony/config
```

If you install this component outside of a Symfony application, you must require the `vendor/autoload.php` file in your code to enable the class autoloading mechanism provided by Composer. Read *this article* for more details.

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

Chapter 19
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 19-1

```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 = [];

    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. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/ConfigCache.php

```
}

// 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 20

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 20-1

```yaml
database:
    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 20-2
```php
namespace Acme\DatabaseConfiguration;

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

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

        // ... add node definitions to the root of the tree
        // $treeBuilder->getRootNode()->...

        return $treeBuilder;
    }
}
```

Deprecated since version 4.2: Not passing the root node name to `TreeBuilder` was deprecated in Symfony 4.2.

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 20-3
```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)

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/TreeBuilder.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/TreeBuilder.php
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/ConfigurationInterface.php

- array
- variable (no validation)

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

Numeric Node Constraints

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

Listing 20-4
```
$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 20-5
```
$rootNode
    ->children()
        ->enumNode('delivery')
            ->values(['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 20-6
```
$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 20-7

4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/IntegerNodeDefinition.php
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/IntegerNodeDefinition.php

```
$rootNode
    ->children()
        ->arrayNode('connections')
            ->arrayPrototype()
                ->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 20-8
```
->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 20-9
```
$node
    ->fixXmlConfig('driver')
    ->children()
        ->arrayNode('drivers')
            ->scalarPrototype()->end()
        ->end()
    ->end()
;
```

When using the following YAML configuration:

Listing 20-10
```yaml
drivers: ['mysql', 'sqlite']
```

Or the following XML configuration:

Listing 20-11
```xml
<driver>mysql</driver>
<driver>sqlite</driver>
```

The processed configuration is:

Listing 20-12
```
Array(
    [0] => 'mysql'
    [1] => 'sqlite'
)
```

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

Listing 20-13
```
$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 20-14
```yaml
connections:
    - { table: symfony, user: root, password: ~ }
    - { table: foo, user: root, password: pa$$ }
```

Or the following XML configuration:

Listing 20-15
```xml
<connection table="symfony" user="root" password="null"/>
<connection table="foo" user="root" password="pa$$"/>
```

The processed configuration is:

Listing 20-16
```
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 20-17
```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 20-18
```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 20-19
```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 20-20
```
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 `*`.

The following example shows these methods in practice:

Listing 20-21
```php
$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 20-22
```php
$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.')
```

6. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/NodeDefinition.php

```
            ->end()
        ->end()
    ;
```

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

Documenting the Option

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

Listing 20-23
```
$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 20-24
```
# This value is only used for the search results page.
entries_per_page: 25
```

and in XML:

Listing 20-25
```
<!-- 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 20-26
```
$arrayNode
    ->canBeEnabled()
;

// is equivalent to

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

7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/NodeDefinition.php
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.php
9. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.php

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

Merging Options

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

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

For all nodes:

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

Appending Sections

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

Listing 20-27
```php
public function getConfigTreeBuilder()
{
    $treeBuilder = new TreeBuilder('database');

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

    return $treeBuilder;
}

public function addParametersNode()
{
    $treeBuilder = new TreeBuilder('parameters');

    $node = $treeBuilder->getRootNode()
        ->isRequired()
        ->requiresAtLeastOneElement()
        ->useAttributeAsKey('name')
        ->arrayPrototype()
            ->children()
                ->scalarNode('value')->isRequired()->end()
            ->end()
        ->end()
```

```
43      ;
44
45      return $node;
46  }
```

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 20-28
```
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 20-29
```
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 20-30
```
<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 20-31
```
$rootNode
    ->fixXmlConfig('extension')
    ->children()
        ->arrayNode('extensions')
            ->scalarPrototype()->end()
        ->end()
```

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

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

Listing 20-32
```
$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 20-33
```
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 20-34
```
<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 20-35
```
connection:
    name:     my_mysql_connection
    host:     localhost
    driver:   mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 20-36
```
connection: my_mysql_connection
```

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

Listing 20-37
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
                ->ifString()
                ->then(function ($v) { return ['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 20-38
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->validate()
                        ->ifNotInArray(['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.

Configuring the Node Path Separator

Consider the following config builder example:

Listing 20-39
```
$treeBuilder = new TreeBuilder('database');

$treeBuilder->getRootNode()
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')->end()
            ->end()
        ->end()
    ->end()
;
```

10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/ExprBuilder.php

By default, the hierarchy of nodes in a config path is defined with a dot character (`.`):

Listing 20-40
```
// ...

$node = $treeBuilder->buildTree();
$children = $node->getChildren();
$path = $children['driver']->getPath();
// $path = 'database.connection.driver'
```

Use the `setPathSeparator()` method on the config builder to change the path separator:

Listing 20-41
```
// ...

$treeBuilder->setPathSeparator('/');
$node = $treeBuilder->buildTree();
$children = $node->getChildren();
$path = $children['driver']->getPath();
// $path = 'database/connection/driver'
```

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 20-42
```
use Acme\DatabaseConfiguration;
use Symfony\Component\Config\Definition\Processor;
use Symfony\Component\Yaml\Yaml;

$config = Yaml::parse(
    file_get_contents(__DIR__.'/src/Matthias/config/config.yaml')
);
$extraConfig = Yaml::parse(
    file_get_contents(__DIR__.'/src/Matthias/config/config_extra.yaml')
);

$configs = [$config, $extraConfig];

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

11. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Processor.php
12. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Definition/Builder/TreeBuilder.php

Chapter 21
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 21-1
```php
use Symfony\Component\Config\FileLocator;

$configDirectories = [__DIR__.'/config'];

$fileLocator = new FileLocator($configDirectories);
$yamlUserFiles = $fileLocator->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. https://secure.php.net/manual/en/function.parse-ini-file.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/FileLocator.php
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/LoaderInterface.php
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/FileLoader.php

Listing 21-2
```php
namespace Acme\Config\Loader;

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 21-3
```php
use Acme\Config\Loader\YamlUserLoader;
use Symfony\Component\Config\Loader\DelegatingLoader;
use Symfony\Component\Config\Loader\LoaderResolver;

$loaderResolver = new LoaderResolver([new YamlUserLoader($fileLocator)]);
$delegatingLoader = new DelegatingLoader($loaderResolver);

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

5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/LoaderResolver.php
6. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/DelegatingLoader.php
7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/LoaderResolver.php
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Config/Loader/LoaderResolver.php