Symfony

The Components Book

Version: 2.3

generated on May 4, 2015

The Components Book (2.3)

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

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

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

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

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

Contents at a Glance

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

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

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

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

3. Write your code!

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

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

Listing 1-2
```php
// File example: src/script.php

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

use Symfony\Component\Finder\Finder;

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

// ...
```

Using all of the Components

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

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

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

Now what?

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

And have fun!

Chapter 2
The ClassLoader Component

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

Usage

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

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

Additionally, the Symfony ClassLoader component ships with a set of wrapper classes which can be used to add additional functionality on top of existing autoloaders:

- *Cache a Class Loader*
- *Debugging a Class Loader*

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/class-loader` on *Packagist*[2]);
- Use the official Git repository (*https://github.com/symfony/ClassLoader*[3]).

1. `http://php.net/manual/en/language.oop5.autoload.php`
2. `https://packagist.org/packages/symfony/class-loader`
3. `https://github.com/symfony/ClassLoader`

Chapter 3
The PSR-0 Class Loader

New in version 2.1: The `ClassLoader` class was introduced in Symfony 2.1.

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

You can use both the `ApcClassLoader` and the `XcacheClassLoader` to *cache* a `ClassLoader` instance or the `DebugClassLoader` to *debug* it.

Usage

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

Listing 3-1

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

use Symfony\Component\ClassLoader\ClassLoader;

$loader = new ClassLoader();

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

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

$loader->register();
```

1. http://www.php-fig.org/psr/psr-0/
2. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassLoader.html
3. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassLoader.html

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

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

Listing 3-2
```php
// register a single namespaces
$loader->addPrefix('Symfony', __DIR__.'/vendor/symfony/symfony/src');

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

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

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

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

Listing 3-3
```php
$loader->addPrefixes(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine/common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine/migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine/dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/orm/lib',
));
```

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

4. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassLoader.html#addPrefix()
5. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassLoader.html#addPrefixes()
6. http://pear.php.net/manual/en/standards.naming.php

Chapter 4
MapClassLoader

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

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

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

Usage

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

Listing 4-1

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

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

$loader = new MapClassLoader($mapping);

$loader->register();
```

1. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/MapClassLoader.html
2. http://www.php-fig.org/psr/psr-0/

Chapter 5
Cache a Class Loader

Introduction

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

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

ApcClassLoader

New in version 2.1: The `ApcClassLoader` class was introduced in Symfony 2.1.

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

Listing 5-1
```php
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcClassLoader.php';

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

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

// register the cached class loader
```

1. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ApcClassLoader.html
2. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/XcacheClassLoader.html
3. http://getcomposer.org/doc/01-basic-usage.md#autoloading
4. http://php.net/manual/en/book.apc.php

```
10 $cachedLoader->register();
11
12 // deactivate the original, non-cached loader if it was registered previously
13 $loader->unregister();
```

XcacheClassLoader

New in version 2.1: The `XcacheClassLoader` class was introduced in Symfony 2.1.

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

Listing 5-2

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

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

Chapter 6
Debugging a Class Loader

New in version 2.1: The `DebugClassLoader` class was introduced in Symfony 2.1.

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

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

Listing 6-1
```
1 use Symfony\Component\ClassLoader\DebugClassLoader;
2
3 DebugClassLoader::enable();
```

1. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/DebugClassLoader.html
2. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/DebugClassLoader.html#enable()

Chapter 7
The Class Map Generator

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

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

Listing 7-1

```
library/
├── bar/
│   ├── baz/
│   │   └── Boo.php
│   └── Foo.php
└── foo/
    ├── bar/
    │   └── Foo.php
    └── Bar.php
```

These files contain the following classes:

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

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

---

1. http://www.php-fig.org/psr/psr-0
2. http://www.php-fig.org/psr/psr-4
3. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassMapGenerator.html

Generating a Class Map

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

Listing 7-2
```php
use Symfony\Component\ClassLoader\ClassMapGenerator;

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

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

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

Dumping the Class Map

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

Listing 7-4
```php
use Symfony\Component\ClassLoader\ClassMapGenerator;

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

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

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

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

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

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

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

```

4. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassMapGenerator.html#createMap()
5. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/ClassMapGenerator.html#dump()
6. http://api.symfony.com/2.3/Symfony/Component/ClassLoader/MapClassLoader.html

```
10 $foo = new Foo();
11
12 // ...
```

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

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

Listing 7-7
```
1 use Symfony\Component\ClassLoader\ClassMapGenerator;
2
3 ClassMapGenerator::dump(
4     array(__DIR__.'/library/bar', __DIR__.'/library/foo'),
5     __DIR__.'/class_map.php'
6 );
```

7. http://getcomposer.org

Chapter 8
The Config Component

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

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/config` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Config*[2]).

Sections

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

1. `https://packagist.org/packages/symfony/config`
2. `https://github.com/symfony/Config`

Chapter 9
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 – in most cases: files. This can be done with the *`FileLocator`*[2]:

Listing 9-1
```php
use Symfony\Component\Config\FileLocator;

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

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

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

1. http://php.net/manual/en/function.parse-ini-file.php
2. http://api.symfony.com/2.3/Symfony/Component/Config/FileLocator.html

Resource Loaders

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

Listing 9-2

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

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

        // ... handle the config values

        // maybe import some other resource:

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

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

Finding the right Loader

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

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

Listing 9-3

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

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

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

3. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html
4. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/FileLoader.html
5. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderResolver.html
6. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/DelegatingLoader.html
7. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderResolver.html
8. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderResolver.html

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

Chapter 10
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 10-1

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

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

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

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

    $resources = array();

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

1. http://api.symfony.com/2.3/Symfony/Component/Config/ConfigCache.html

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

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

Chapter 11
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 11-1

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

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

Defining a Hierarchy of Configuration Values Using the TreeBuilder

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

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

Listing 11-2
```php
namespace Acme\DatabaseConfiguration;

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

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

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

        return $treeBuilder;
    }
}
```

Adding Node Definitions to the Tree

Variable Nodes

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

Listing 11-3
```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:

1. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
2. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
3. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/ConfigurationInterface.html

- scalar (generic type that includes booleans, strings, integers, floats and `null`)
- boolean
- integer (new in 2.2)
- float (new in 2.2)
- enum (new in 2.1) (similar to scalar, but it only allows a finite set of values)
- array
- variable (no validation)

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

Numeric Node Constraints

New in version 2.2: The numeric (float and integer) nodes were introduced in Symfony 2.2.

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

Listing 11-4
```php
$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

New in version 2.1: The enum node was introduced in Symfony 2.1.

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

Listing 11-5
```php
$rootNode
    ->children()
        ->enumNode('gender')
            ->values(array('male', 'female'))
        ->end()
    ->end()
;
```

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

Array Nodes

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

Listing 11-6

4. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder.html#min()
5. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder.html#max()

```
$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 11-7
```
$rootNode
    ->children()
        ->arrayNode('connections')
            ->prototype('array')
                ->children()
                    ->scalarNode('driver')->end()
                    ->scalarNode('host')->end()
                    ->scalarNode('username')->end()
                    ->scalarNode('password')->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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

Array Node Options

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

`useAttributeAsKey()`
: Provide the name of a child node, whose value should be used as the key in the resulting array.

`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.

An example of this:

Listing 11-8
```
$rootNode
    ->children()
        ->arrayNode('parameters')
            ->isRequired()
            ->requiresAtLeastOneElement()
            ->useAttributeAsKey('name')
            ->prototype('array')
                ->children()
```

```
                    ->scalarNode('value')->isRequired()->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

In YAML, the configuration might look like this:

Listing 11-9
```
database:
    parameters:
        param1: { value: param1val }
```

In XML, each `parameters` node would have a `name` attribute (along with `value`), which would be removed and used as the key for that element in the final array. The `useAttributeAsKey` is useful for normalizing how arrays are specified between different formats like XML and YAML.

Default and required Values

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

`defaultValue()`
: Set a default value

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

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

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

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

Listing 11-10
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->cannotBeEmpty()
                ->end()
                ->scalarNode('host')
                    ->defaultValue('localhost')
                ->end()
                ->scalarNode('username')->end()
                ->scalarNode('password')->end()
                ->booleanNode('memory')
                    ->defaultFalse()
                ->end()
            ->end()
```

```
        ->end()
        ->arrayNode('settings')
            ->addDefaultsIfNotSet()
            ->children()
                ->scalarNode('name')
                    ->isRequired()
                    ->cannotBeEmpty()
                    ->defaultValue('value')
                ->end()
            ->end()
        ->end()
    ->end()
;
```

Documenting the Option

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

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

Optional Sections

New in version 2.2: The `canBeEnabled` and `canBeDisabled` methods were introduced in Symfony 2.2.

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

Listing 11-11
```
$arrayNode
    ->canBeEnabled()
;

// is equivalent to

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

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

Merging Options

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

6. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/NodeDefinition.html#info()
7. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#canBeEnabled()
8. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/ArrayNodeDefinition.html#canBeDisabled()

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

For all nodes:

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

Appending Sections

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

Listing 11-12

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

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

    return $treeBuilder;
}

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

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

```
            ->end()
        ;

        return $node;
    }
```

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

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 11-13
```yaml
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 11-14
```xml
<twig:config>
    <twig:extension>twig.extension.foo</twig:extension>
    <twig:extension>twig.extension.bar</twig:extension>
</twig:config>
```

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

Listing 11-15
```
$rootNode
    ->fixXmlConfig('extension')
    ->children()
        ->arrayNode('extensions')
            ->prototype('scalar')->end()
        ->end()
    ->end()
;
```

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

Listing 11-16
```
$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 11-17
```
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 11-18
```
<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 11-19
```
connection:
    name:     my_mysql_connection
    host:     localhost
    driver:   mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 11-20
```
connection: my_mysql_connection
```

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

Listing 11-21
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
                ->ifString()
                ->then(function ($v) { return array('name' => $v); })
            ->end()
            ->children()
                ->scalarNode('name')->isRequired()
                // ...
            ->end()
        ->end()
    ->end()
;
```

Validation Rules

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

Listing 11-22
```php
$rootNode
    ->children()
        ->arrayNode('connection')
            ->children()
                ->scalarNode('driver')
                    ->isRequired()
                    ->validate()
                    ->ifNotInArray(array('mysql', 'sqlite', 'mssql'))
                        ->thenInvalid('Invalid database driver "%s"')
                    ->end()
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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

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

A validation rule also requires a "then" part:

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

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

Processing Configuration Values

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

Listing 11-23
```php
use Symfony\Component\Yaml\Yaml;
use Symfony\Component\Config\Definition\Processor;
```

9. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/ExprBuilder.html
10. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Processor.html
11. http://api.symfony.com/2.3/Symfony/Component/Config/Definition/Builder/TreeBuilder.html

```php
use Acme\DatabaseConfiguration;

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

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

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

Chapter 12
The Console Component

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

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

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/console` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Console*[2]).

Creating a basic Command

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

Listing 12-1

```
1  namespace Acme\Console\Command;
2
3  use Symfony\Component\Console\Command\Command;
4  use Symfony\Component\Console\Input\InputArgument;
5  use Symfony\Component\Console\Input\InputInterface;
6  use Symfony\Component\Console\Input\InputOption;
7  use Symfony\Component\Console\Output\OutputInterface;
8
9  class GreetCommand extends Command
10 {
```

1. https://packagist.org/packages/symfony/console
2. https://github.com/symfony/Console

```
    protected function configure()
    {
        $this
            ->setName('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument(
                'name',
                InputArgument::OPTIONAL,
                'Who do you want to greet?'
            )
            ->addOption(
               'yell',
               null,
               InputOption::VALUE_NONE,
               'If set, the task will yell in uppercase letters'
            )
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $name = $input->getArgument('name');
        if ($name) {
            $text = 'Hello '.$name;
        } else {
            $text = 'Hello';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }

        $output->writeln($text);
    }
}
```

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

Listing 12-2
```
#!/usr/bin/env php
<?php
// application.php

require __DIR__.'/vendor/autoload.php';

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

$application = new Application();
$application->add(new GreetCommand());
$application->run();
```

Test the new console command by running the following

Listing 12-3
```
$ php application.php demo:greet Fabien
```

This will print the following to the command line:

Listing 12-4
```
1 Hello Fabien
```

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

Listing 12-5
```
1 $ php application.php demo:greet Fabien --yell
```

This prints:

Listing 12-6
```
1 HELLO FABIEN
```

Coloring the Output

By default, the Windows command console doesn't support output coloring. The Console component disables output coloring for Windows systems, but if your commands invoke other scripts which emit color sequences, they will be wrongly displayed as raw escape characters. Install the *ConEmu*[3] or *ANSICON*[4] free applications to add coloring support to your Windows command console.

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

Listing 12-7
```
1  // green text
2  $output->writeln('<info>foo</info>');
3
4  // yellow text
5  $output->writeln('<comment>foo</comment>');
6
7  // black text on a cyan background
8  $output->writeln('<question>foo</question>');
9
10 // white text on a red background
11 $output->writeln('<error>foo</error>');
```

It is possible to define your own styles using the class *`OutputFormatterStyle`*[5]:

Listing 12-8
```
1 use Symfony\Component\Console\Formatter\OutputFormatterStyle;
2
3 // ...
4 $style = new OutputFormatterStyle('red', 'yellow', array('bold', 'blink'));
5 $output->getFormatter()->setStyle('fire', $style);
6 $output->writeln('<fire>foo</fire>');
```

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

And available options are: `bold`, `underscore`, `blink`, `reverse` and `conceal`.

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

Listing 12-9

3. https://code.google.com/p/conemu-maximus5/
4. https://github.com/adoxa/ansicon/releases
5. http://api.symfony.com/2.3/Symfony/Component/Console/Formatter/OutputFormatterStyle.html

```php
// green text
$output->writeln('<fg=green>foo</fg=green>');

// black text on a cyan background
$output->writeln('<fg=black;bg=cyan>foo</fg=black;bg=cyan>');

// bold text on a yellow background
$output->writeln('<bg=yellow;options=bold>foo</bg=yellow;options=bold>');
```

Verbosity Levels

New in version 2.3: The `VERBOSITY_VERY_VERBOSE` and `VERBOSITY_DEBUG` constants were introduced in version 2.3

The console has 5 levels of verbosity. These are defined in the *`OutputInterface`*[6]:

Mode	Value
OutputInterface::VERBOSITY_QUIET	Do not output any messages
OutputInterface::VERBOSITY_NORMAL	The default verbosity level
OutputInterface::VERBOSITY_VERBOSE	Increased verbosity of messages
OutputInterface::VERBOSITY_VERY_VERBOSE	Informative non essential messages
OutputInterface::VERBOSITY_DEBUG	Debug messages

You can specify the quiet verbosity level with the `--quiet` or `-q` option. The `--verbose` or `-v` option is used when you want an increased level of verbosity.

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

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

Listing 12-10
```php
if (OutputInterface::VERBOSITY_VERBOSE <= $output->getVerbosity()) {
    $output->writeln(...);
}
```

When the quiet level is used, all output is suppressed as the default *`write()`*[7] method returns without actually printing.

Using Command Arguments

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

Listing 12-11

6. http://api.symfony.com/2.3/Symfony/Component/Console/Output/OutputInterface.html
7. http://api.symfony.com/2.3/Symfony/Component/Console/Output/Output.html#write()

```
$this
    // ...
    ->addArgument(
        'name',
        InputArgument::REQUIRED,
        'Who do you want to greet?'
    )
    ->addArgument(
        'last_name',
        InputArgument::OPTIONAL,
        'Your last name?'
    );
```

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

Listing 12-12
```
if ($lastName = $input->getArgument('last_name')) {
    $text .= ' '.$lastName;
}
```

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

Listing 12-13
```
$ php application.php demo:greet Fabien
$ php application.php demo:greet Fabien Potencier
```

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

Listing 12-14
```
$this
    // ...
    ->addArgument(
        'names',
        InputArgument::IS_ARRAY,
        'Who do you want to greet (separate multiple names with a space)?'
    );
```

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

Listing 12-15
```
$ php application.php demo:greet Fabien Ryan Bernhard
```

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

Listing 12-16
```
if ($names = $input->getArgument('names')) {
    $text .= ' '.implode(', ', $names);
}
```

There are 3 argument variants you can use:

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

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

Listing 12-17
```
$this
    // ...
    ->addArgument(
        'names',
        InputArgument::IS_ARRAY | InputArgument::REQUIRED,
        'Who do you want to greet (separate multiple names with a space)?'
    );
```

Using Command Options

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

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

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

Listing 12-18
```
$this
    // ...
    ->addOption(
        'iterations',
        null,
        InputOption::VALUE_REQUIRED,
        'How many times should the message be printed?',
        1
    );
```

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

Listing 12-19
```
for ($i = 0; $i < $input->getOption('iterations'); $i++) {
    $output->writeln($text);
}
```

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

Listing 12-20
```
$ php application.php demo:greet Fabien
$ php application.php demo:greet Fabien --iterations=5
```

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

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

Listing 12-21

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

There are 4 option variants you can use:

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

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

Listing 12-22
```
$this
    // ...
    ->addOption(
        'colors',
        null,
        InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY,
        'Which colors do you like?',
        array('blue', 'red')
    );
```

Console Helpers

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

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

Testing Commands

Symfony provides several tools to help you test your commands. The most useful one is the *`CommandTester`*[8] class. It uses special input and output classes to ease testing without a real console:

Listing 12-23
```
use Acme\Console\Command\GreetCommand;
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
```

8. http://api.symfony.com/2.3/Symfony/Component/Console/Tester/CommandTester.html

```
    public function testExecute()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getName()));

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

        // ...
    }
}
```

The *getDisplay()*[9] method returns what would have been displayed during a normal call from the console.

You can test sending arguments and options to the command by passing them as an array to the *execute()*[10] method:

Listing 12-24

```
use Acme\Console\Command\GreetCommand;
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    // ...

    public function testNameIsOutput()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array(
            'command'      => $command->getName(),
            'name'         => 'Fabien',
            '--iterations' => 5,
        ));

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

You can also test a whole console application by using *ApplicationTester*[11].

9. http://api.symfony.com/2.3/Symfony/Component/Console/Tester/CommandTester.html#getDisplay()
10. http://api.symfony.com/2.3/Symfony/Component/Console/Tester/CommandTester.html#execute()
11. http://api.symfony.com/2.3/Symfony/Component/Console/Tester/ApplicationTester.html

Calling an Existing Command

If a command depends on another one being run before it, instead of asking the user to remember the order of execution, you can call it directly yourself. This is also useful if you want to create a "meta" command that just runs a bunch of other commands (for instance, all commands that need to be run when the project's code has changed on the production servers: clearing the cache, generating Doctrine2 proxies, dumping Assetic assets, ...).

Calling a command from another one is straightforward:

Listing 12-25
```php
protected function execute(InputInterface $input, OutputInterface $output)
{
    $command = $this->getApplication()->find('demo:greet');

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

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

    // ...
}
```

First, you *`find()`*[12] the command you want to execute by passing the command name.

Then, you need to create a new *`ArrayInput`*[13] with the arguments and options you want to pass to the command.

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

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

Learn More!

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

12. http://api.symfony.com/2.3/Symfony/Component/Console/Application.html#find()
13. http://api.symfony.com/2.3/Symfony/Component/Console/Input/ArrayInput.html

Chapter 13
Using Console Commands, Shortcuts and Built-in Commands

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

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

Listing 13-1
```php
#!/usr/bin/env php
<?php
// application.php

use Symfony\Component\Console\Application;

$application = new Application();
// ...
$application->run();
```

Built-in Commands

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

Listing 13-2
```
$ php application.php list
```

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

Listing 13-3
```
$ php application.php
```

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

Listing 13-4
```
$ php application.php help list
```

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

Listing 13-5
```
$ php application.php help
```

Global Options

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

Listing 13-6
```
$ php application.php list --help
$ php application.php list -h
```

You can suppress output with:

Listing 13-7
```
$ php application.php list --quiet
$ php application.php list -q
```

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

Listing 13-8
```
$ php application.php list --verbose
$ php application.php list -v
```

The verbose flag can optionally take a value between 1 (default) and 3 to output even more verbose messages:

Listing 13-9
```
$ php application.php list --verbose=2
$ php application.php list -vv
$ php application.php list --verbose=3
$ php application.php list -vvv
```

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

Listing 13-10
```
$application = new Application('Acme Console Application', '1.2');
```

then you can use:

Listing 13-11
```
$ php application.php list --version
$ php application.php list -V
```

to get this information output:

Listing 13-12
```
Acme Console Application version 1.2
```

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

Listing 13-13
```
console tool
```

You can force turning on ANSI output coloring with:

Listing 13-14
```
$ php application.php list --ansi
```

or turn it off with:

Listing 13-15
```
$ php application.php list --no-ansi
```

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

Listing 13-16
```
$ php application.php list --no-interaction
$ php application.php list -n
```

Shortcut Syntax

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

Listing 13-17
```
$ php application.php h
```

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

Listing 13-18
```
$ php application.php d:g Fabien
```

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

Chapter 14
Building a single Command Application

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

Listing 14-1
```php
namespace Acme\Tool;

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

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

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

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

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

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

You can also simplify how you execute the application:

Listing 14-2
```
1  #!/usr/bin/env php
2  <?php
3  // command.php
4
5  use Acme\Tool\MyApplication;
6
7  $application = new MyApplication();
8  $application->run();
```

Chapter 15

Understanding how Console Arguments Are Handled

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

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

Listing 15-1

```php
namespace Acme\Console\Command;

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

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

    protected function execute(InputInterface $input, OutputInterface $output)
```

1. http://docopt.org/

```
26      {
27          // ...
28      }
29  }
```

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

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

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

Listing 15-2
```
// ...

new InputDefinition(array(
    // ...
    new InputArgument('arg', InputArgument::OPTIONAL),
));
```

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

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

Chapter 16
Using Events

New in version 2.3: Console events were introduced in Symfony 2.3.

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

Listing 16-1
```php
use Symfony\Component\Console\Application;
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();

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

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

The `ConsoleEvents::COMMAND` Event

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

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

Listing 16-2
```php
use Symfony\Component\Console\Event\ConsoleCommandEvent;
use Symfony\Component\Console\ConsoleEvents;

```

1. http://api.symfony.com/2.3/Symfony/Component/Console/Event/ConsoleCommandEvent.html

```
$dispatcher->addListener(ConsoleEvents::COMMAND, function (ConsoleCommandEvent $event) {
    // get the input instance
    $input = $event->getInput();

    // get the output instance
    $output = $event->getOutput();

    // get the command to be executed
    $command = $event->getCommand();

    // write something about the command
    $output->writeln(sprintf('Before running command <info>%s</info>',
$command->getName()));

    // get the application
    $application = $command->getApplication();
});
```

The `ConsoleEvents::TERMINATE` Event

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

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

Listeners receive a *`ConsoleTerminateEvent`*[2] event:

Listing 16-3
```
use Symfony\Component\Console\Event\ConsoleTerminateEvent;
use Symfony\Component\Console\ConsoleEvents;

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

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

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

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

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

2. http://api.symfony.com/2.3/Symfony/Component/Console/Event/ConsoleTerminateEvent.html

The `ConsoleEvents::EXCEPTION` Event

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

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

Listeners receive a *`ConsoleExceptionEvent`*[3] event:

Listing 16-4

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

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

    $command = $event->getCommand();

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

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

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

3. http://api.symfony.com/2.3/Symfony/Component/Console/Event/ConsoleExceptionEvent.html

Chapter 17
Dialog Helper

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

Listing 17-1
```php
$dialog = $this->getHelper('dialog');
```

All the methods inside the Dialog Helper have an *OutputInterface*[3] as the first argument, the question as the second argument and the default value as the last argument.

Asking the User for Confirmation

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

Listing 17-2
```php
// ...
if (!$dialog->askConfirmation(
        $output,
        '<question>Continue with this action?</question>',
        false
    )) {
    return;
}
```

In this case, the user will be asked "Continue with this action?", and will return `true` if the user answers with `y` or `false` if the user answers with `n`. The third argument to *askConfirmation()*[4] is the default value to return if the user doesn't enter any input. Any other input will ask the same question again.

1. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html
2. http://api.symfony.com/2.3/Symfony/Component/Console/Command/Command.html#getHelperSet()
3. http://api.symfony.com/2.3/Symfony/Component/Console/Output/OutputInterface.html
4. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html#askConfirmation()

Asking the User for Information

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

Listing 17-3
```
// ...
$bundle = $dialog->ask(
    $output,
    'Please enter the name of the bundle',
    'AcmeDemoBundle'
);
```

The user will be asked "Please enter the name of the bundle". They can type some name which will be returned by the *ask()*[5] method. If they leave it empty, the default value (AcmeDemoBundle here) is returned.

Autocompletion

New in version 2.2: Autocompletion for questions was introduced in Symfony 2.2.

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

Listing 17-4
```
$dialog = $this->getHelper('dialog');
$bundleNames = array('AcmeDemoBundle', 'AcmeBlogBundle', 'AcmeStoreBundle');
$name = $dialog->ask(
    $output,
    'Please enter the name of a bundle',
    'FooBundle',
    $bundleNames
);
```

Hiding the User's Response

New in version 2.2: The `askHiddenResponse` method was introduced in Symfony 2.2.

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

Listing 17-5
```
$dialog = $this->getHelper('dialog');
$password = $dialog->askHiddenResponse(
    $output,
    'What is the database password?',
    false
);
```

When you ask for a hidden response, Symfony will use either a binary, change stty mode or use another trick to hide the response. If none is available, it will fallback and allow the response to be visible unless you pass `false` as the third argument like in the example above. In this case, a RuntimeException would be thrown.

5. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html#ask()

Validating the Answer

You can even validate the answer. For instance, in the last example you asked for the bundle name. Following the Symfony naming conventions, it should be suffixed with `Bundle`. You can validate that by using the *`askAndValidate()`*[6] method:

Listing 17-6
```php
// ...
$bundle = $dialog->askAndValidate(
    $output,
    'Please enter the name of the bundle',
    function ($answer) {
        if ('Bundle' !== substr($answer, -6)) {
            throw new \RuntimeException(
                'The name of the bundle should be suffixed with \'Bundle\''
            );
        }

        return $answer;
    },
    false,
    'AcmeDemoBundle'
);
```

This methods has 2 new arguments, the full signature is:

Listing 17-7
```
askAndValidate(
    OutputInterface $output,
    string|array $question,
    callback $validator,
    integer $attempts = false,
    string $default = null,
    array $autocomplete = null
)
```

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

You can set the max number of times to ask in the `$attempts` argument. If you reach this max number it will use the default value. Using `false` means the amount of attempts is infinite. The user will be asked as long as they provide an invalid answer and will only be able to proceed if their input is valid.

Validating a Hidden Response

New in version 2.2: The `askHiddenResponseAndValidate` method was introduced in Symfony 2.2.

You can also ask and validate a hidden response:

Listing 17-8
```php
$dialog = $this->getHelper('dialog');

$validator = function ($value) {
    if ('' === trim($value)) {
        throw new \Exception('The password can not be empty');
    }
```

6. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html#askAndValidate()

```
7
8       return $value;
9   };
10
11  $password = $dialog->askHiddenResponseAndValidate(
12      $output,
13      'Please enter your password',
14      $validator,
15      20,
16      false
17  );
```

If you want to allow the response to be visible if it cannot be hidden for some reason, pass true as the fifth argument.

Let the User Choose from a List of Answers

New in version 2.2: The ***select()***[7] method was introduced in Symfony 2.2.

If you have a predefined set of answers the user can choose from, you could use the `ask` method described above or, to make sure the user provided a correct answer, the `askAndValidate` method. Both have the disadvantage that you need to handle incorrect values yourself.

Instead, you can use the ***select()***[8] method, which makes sure that the user can only enter a valid string from a predefined list:

Listing 17-9

```
1  $dialog = $this->getHelper('dialog');
2  $colors = array('red', 'blue', 'yellow');
3
4  $color = $dialog->select(
5      $output,
6      'Please select your favorite color (default to red)',
7      $colors,
8      0
9  );
10 $output->writeln('You have just selected: ' . $colors[$color]);
11
12 // ... do something with the color
```

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

If the user enters an invalid string, an error message is shown and the user is asked to provide the answer another time, until they enter a valid string or the maximum attempts is reached (which you can define in the fifth argument). The default value for the attempts is `false`, which means infinite attempts. You can define your own error message in the sixth argument.

New in version 2.3: Multiselect support was introduced in Symfony 2.3.

Multiple Choices

Sometimes, multiple answers can be given. The DialogHelper provides this feature using comma separated values. This is disabled by default, to enable this set the seventh argument to `true`:

Listing 17-10

7. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html#select()
8. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/DialogHelper.html#select()

```php
// ...

$selected = $dialog->select(
    $output,
    'Please select your favorite color (default to red)',
    $colors,
    0,
    false,
    'Value "%s" is invalid',
    true // enable multiselect
);

$selectedColors = array_map(function ($c) use ($colors) {
    return $colors[$c];
}, $selected);

$output->writeln(
    'You have just selected: ' . implode(', ', $selectedColors)
);
```

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

Testing a Command which Expects Input

If you want to write a unit test for a command which expects some kind of input from the command line, you need to overwrite the HelperSet used by the command:

Listing 17-11

```php
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Helper\DialogHelper;
use Symfony\Component\Console\Helper\HelperSet;
use Symfony\Component\Console\Tester\CommandTester;

// ...
public function testExecute()
{
    // ...
    $application = new Application();
    $application->add(new MyCommand());
    $command = $application->find('my:command:name');
    $commandTester = new CommandTester($command);

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

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

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

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

```
30
31      return $stream;
32  }
```

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

> *You find more information about testing commands in the console component docs about testing console commands.*

Chapter 18
Formatter Helper

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

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

Listing 18-1
```
$formatter = $this->getHelper('formatter');
```

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

Print Messages in a Section

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

Listing 18-2
```
[SomeSection] Here is some message related to that section
```

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

Listing 18-3
```
$formattedLine = $formatter->formatSection(
    'SomeSection',
    'Here is some message related to that section'
);
$output->writeln($formattedLine);
```

1. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/FormatterHelper.html
2. http://api.symfony.com/2.3/Symfony/Component/Console/Command/Command.html#getHelperSet()
3. http://api.symfony.com/2.3/Symfony/Component/Console/Output/OutputInterface.html#writeln()
4. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/FormatterHelper.html#formatSection()

Print Messages in a Block

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

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

Listing 18-4
```
$errorMessages = array('Error!', 'Something went wrong');
$formattedBlock = $formatter->formatBlock($errorMessages, 'error');
$output->writeln($formattedBlock);
```

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

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

5. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/FormatterHelper.html#formatBlock()
6. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/FormatterHelper.html#formatBlock()

Chapter 19
Progress Helper

New in version 2.2: The `progress` helper was introduced in Symfony 2.2.

New in version 2.3: The `setCurrent` method was introduced in Symfony 2.3.

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

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

Listing 19-1
```php
$progress = $this->getHelper('progress');

$progress->start($output, 50);
$i = 0;
while ($i++ < 50) {
    // ... do some work

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

$progress->finish();
```

You can also set the current progress by calling the *setCurrent()*[2] method.

1. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/ProgressHelper.html
2. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/ProgressHelper.html#setCurrent()

The appearance of the progress output can be customized as well, with a number of different levels of verbosity. Each of these displays different possible items - like percentage completion, a moving progress bar, or current/total information (e.g. 10/50):

Listing 19-2
```
$progress->setFormat(ProgressHelper::FORMAT_QUIET);
$progress->setFormat(ProgressHelper::FORMAT_NORMAL);
$progress->setFormat(ProgressHelper::FORMAT_VERBOSE);
$progress->setFormat(ProgressHelper::FORMAT_QUIET_NOMAX);
// the default value
$progress->setFormat(ProgressHelper::FORMAT_NORMAL_NOMAX);
$progress->setFormat(ProgressHelper::FORMAT_VERBOSE_NOMAX);
```

You can also control the different characters and the width used for the progress bar:

Listing 19-3
```
// the finished part of the bar
$progress->setBarCharacter('<comment>=</comment>');
// the unfinished part of the bar
$progress->setEmptyBarCharacter(' ');
$progress->setProgressCharacter('|');
$progress->setBarWidth(50);
```

To see other available options, check the API documentation for *ProgressHelper*[3].

For performance reasons, be careful if you set the total number of steps to a high number. For example, if you're iterating over a large number of items, consider setting the redraw frequency to a higher value by calling *setRedrawFrequency()*[4], so it updates on only some iterations:

Listing 19-4
```
$progress->start($output, 50000);

// update every 100 iterations
$progress->setRedrawFrequency(100);

$i = 0;
while ($i++ < 50000) {
    // ... do some work

    $progress->advance();
}
```

3. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/ProgressHelper.html
4. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/ProgressHelper.html#setRedrawFrequency()

Chapter 20
Table Helper

New in version 2.3: The `table` helper was introduced in Symfony 2.3.

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

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

To display a table, use the *TableHelper*[1], set headers, rows and render:

Listing 20-1
```php
$table = $this->getHelper('table');
$table
    ->setHeaders(array('ISBN', 'Title', 'Author'))
    ->setRows(array(
        array('99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'),
        array('9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'),
        array('960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'),
        array('80-902734-1-6', 'And Then There Were None', 'Agatha Christie'),
    ))
;
$table->render($output);
```

The table layout can be customized as well. There are two ways to customize table rendering: using named layouts or by customizing rendering options.

Customize Table Layout using Named Layouts

The Table helper ships with two preconfigured table layouts:

1. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html

- `TableHelper::LAYOUT_DEFAULT`
- `TableHelper::LAYOUT_BORDERLESS`

Layout can be set using *`setLayout()`*[2] method.

Customize Table Layout using Rendering Options

You can also control table rendering by setting custom rendering option values:

- *`setPaddingChar()`*[3]
- *`setHorizontalBorderChar()`*[4]
- *`setVerticalBorderChar()`*[5]
- *`setCrossingChar()`*[6]
- *`setCellHeaderFormat()`*[7]
- *`setCellRowFormat()`*[8]
- *`setBorderFormat()`*[9]
- *`setPadType()`*[10]

2. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setLayout()
3. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setPaddingChar()
4. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setHorizontalBorderChar()
5. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setVerticalBorderChar()
6. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setCrossingChar()
7. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setCellHeaderFormat()
8. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setCellRowFormat()
9. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setBorderFormat()
10. http://api.symfony.com/2.3/Symfony/Component/Console/Helper/TableHelper.html#setPadType()

Chapter 21
The CssSelector Component

> The CssSelector component converts CSS selectors to XPath expressions.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/css-selector` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/CssSelector*[2]).

Usage

Why to Use CSS selectors?

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

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

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

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

1. `https://packagist.org/packages/symfony/css-selector`
2. `https://github.com/symfony/CssSelector`

The CssSelector Component

The component's only goal is to convert CSS selectors to their XPath equivalents:

Listing 21-1
```php
use Symfony\Component\CssSelector\CssSelector;

print CssSelector::toXPath('div.item > h4 > a');
```

This gives the following output:

Listing 21-2
```
descendant-or-self::div[@class and contains(concat(' ',normalize-space(@class), ' '), '
item ')]/h4/a
```

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

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

Limitations of the CssSelector Component

Not all CSS selectors can be converted to XPath equivalents.

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

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

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

Several pseudo-classes are not yet supported:

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

3. http://php.net/manual/en/class.domxpath.php
4. http://php.net/manual/en/class.simplexmlelement.php
5. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html#filter()

Chapter 22
The Debug Component

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

New in version 2.3: The Debug component was introduced in Symfony 2.3. Previously, the classes were located in the HttpKernel component.

Installation

You can install the component in many different ways:

- *Install it via Composer* (`symfony/debug` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Debug*[2]).

Usage

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

Listing 22-1
```php
use Symfony\Component\Debug\Debug;

Debug::enable();
```

The *`enable()`*[3] method registers an error handler and an exception handler. If the *ClassLoader component* is available, a special class loader is also registered.

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

1. https://packagist.org/packages/symfony/debug
2. https://github.com/symfony/Debug
3. http://api.symfony.com/2.3/Symfony/Component/Debug/Debug.html#enable()

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

Enabling the Error Handler

The *ErrorHandler*[4] class catches PHP errors and converts them to exceptions (of class *ErrorException*[5] or *FatalErrorException*[6] for PHP fatal errors):

Listing 22-2
```
use Symfony\Component\Debug\ErrorHandler;

ErrorHandler::register();
```

Enabling the Exception Handler

The *ExceptionHandler*[7] class catches uncaught PHP exceptions and converts them to a nice PHP response. It is useful in debug mode to replace the default PHP/XDebug output with something prettier and more useful:

Listing 22-3
```
use Symfony\Component\Debug\ExceptionHandler;

ExceptionHandler::register();
```

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

4. http://api.symfony.com/2.3/Symfony/Component/Debug/ErrorHandler.html
5. http://php.net/manual/en/class.errorexception.php
6. http://api.symfony.com/2.3/Symfony/Component/Debug/Exception/FatalErrorException.html
7. http://api.symfony.com/2.3/Symfony/Component/Debug/ExceptionHandler.html

Chapter 23
The DependencyInjection Component

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

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

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/dependency-injection` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/DependencyInjection*[2]).

Basic Usage

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

Listing 23-1

```
1  class Mailer
2  {
3      private $transport;
4
5      public function __construct()
6      {
7          $this->transport = 'sendmail';
8      }
9
10     // ...
11 }
```

1. https://packagist.org/packages/symfony/dependency-injection
2. https://github.com/symfony/DependencyInjection

You can register this in the container as a service:

Listing 23-2
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->register('mailer', 'Mailer');
```

An improvement to the class to make it more flexible would be to allow the container to set the `transport` used. If you change the class so this is passed into the constructor:

Listing 23-3
```php
class Mailer
{
    private $transport;

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

    // ...
}
```

Then you can set the choice of transport in the container:

Listing 23-4
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container
    ->register('mailer', 'Mailer')
    ->addArgument('sendmail');
```

This class is now much more flexible as you have separated the choice of transport out of the implementation and into the container.

Which mail transport you have chosen may be something other services need to know about. You can avoid having to change it in multiple places by making it a parameter in the container and then referring to this parameter for the `Mailer` service's constructor argument:

Listing 23-5
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->setParameter('mailer.transport', 'sendmail');
$container
    ->register('mailer', 'Mailer')
    ->addArgument('%mailer.transport%');
```

Now that the `mailer` service is in the container you can inject it as a dependency of other classes. If you have a `NewsletterManager` class like this:

Listing 23-6
```php
class NewsletterManager
{
    private $mailer;

    public function __construct(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }
```

```
9
10      // ...
11  }
```

Then you can register this as a service as well and pass the `mailer` service into it:

Listing 23-7
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2  use Symfony\Component\DependencyInjection\Reference;
3
4  $container = new ContainerBuilder();
5
6  $container->setParameter('mailer.transport', 'sendmail');
7  $container
8      ->register('mailer', 'Mailer')
9      ->addArgument('%mailer.transport%');
10
11 $container
12     ->register('newsletter_manager', 'NewsletterManager')
13     ->addArgument(new Reference('mailer'));
```

If the `NewsletterManager` did not require the `Mailer` and injecting it was only optional then you could use setter injection instead:

Listing 23-8
```
1  class NewsletterManager
2  {
3      private $mailer;
4
5      public function setMailer(\Mailer $mailer)
6      {
7          $this->mailer = $mailer;
8      }
9
10     // ...
11 }
```

You can now choose not to inject a `Mailer` into the `NewsletterManager`. If you do want to though then the container can call the setter method:

Listing 23-9
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2  use Symfony\Component\DependencyInjection\Reference;
3
4  $container = new ContainerBuilder();
5
6  $container->setParameter('mailer.transport', 'sendmail');
7  $container
8      ->register('mailer', 'Mailer')
9      ->addArgument('%mailer.transport%');
10
11 $container
12     ->register('newsletter_manager', 'NewsletterManager')
13     ->addMethodCall('setMailer', array(new Reference('mailer')));
```

You could then get your `newsletter_manager` service from the container like this:

Listing 23-10
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2
```

```php
$container = new ContainerBuilder();

// ...

$newsletterManager = $container->get('newsletter_manager');
```

Avoiding your Code Becoming Dependent on the Container

Whilst you can retrieve services from the container directly it is best to minimize this. For example, in the `NewsletterManager` you injected the `mailer` service in rather than asking for it from the container. You could have injected the container in and retrieved the `mailer` service from it but it would then be tied to this particular container making it difficult to reuse the class elsewhere.

You will need to get a service from the container at some point but this should be as few times as possible at the entry point to your application.

Setting up the Container with Configuration Files

As well as setting up the services using PHP as above you can also use configuration files. This allows you to use XML or YAML to write the definitions for the services rather than using PHP to define the services as in the above examples. In anything but the smallest applications it makes sense to organize the service definitions by moving them into one or more configuration files. To do this you also need to install *the Config component*.

Loading an XML config file:

Listing 23-11
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;

$container = new ContainerBuilder();
$loader = new XmlFileLoader($container, new FileLocator(__DIR__));
$loader->load('services.xml');
```

Loading a YAML config file:

Listing 23-12
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

$container = new ContainerBuilder();
$loader = new YamlFileLoader($container, new FileLocator(__DIR__));
$loader->load('services.yml');
```

If you want to load YAML config files then you will also need to install *the Yaml component*.

If you *do* want to use PHP to create the services then you can move this into a separate config file and load it in a similar way:

Listing 23-13

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\PhpFileLoader;

$container = new ContainerBuilder();
$loader = new PhpFileLoader($container, new FileLocator(__DIR__));
$loader->load('services.php');
```

You can now set up the `newsletter_manager` and `mailer` services using config files:

Listing 23-14

```yaml
parameters:
    # ...
    mailer.transport: sendmail

services:
    mailer:
        class:     Mailer
        arguments: ["%mailer.transport%"]
    newsletter_manager:
        class:     NewsletterManager
        calls:
            - [setMailer, ["@mailer"]]
```

Chapter 24
Types of Injection

Making a class's dependencies explicit and requiring that they be injected into it is a good way of making a class more reusable, testable and decoupled from others.

There are several ways that the dependencies can be injected. Each injection point has advantages and disadvantages to consider, as well as different ways of working with them when using the service container.

Constructor Injection

The most common way to inject dependencies is via a class's constructor. To do this you need to add an argument to the constructor signature to accept the dependency:

Listing 24-1
```php
class NewsletterManager
{
    protected $mailer;

    public function __construct(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

You can specify what service you would like to inject into this in the service container configuration:

Listing 24-2
```yaml
services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     NewsletterManager
        arguments: ["@my_mailer"]
```

Type hinting the injected object means that you can be sure that a suitable dependency has been injected. By type-hinting, you'll get a clear error immediately if an unsuitable dependency is injected. By type hinting using an interface rather than a class you can make the choice of dependency more flexible. And assuming you only use methods defined in the interface, you can gain that flexibility and still safely use the object.

There are several advantages to using constructor injection:

- If the dependency is a requirement and the class cannot work without it then injecting it via the constructor ensures it is present when the class is used as the class cannot be constructed without it.
- The constructor is only ever called once when the object is created, so you can be sure that the dependency will not change during the object's lifetime.

These advantages do mean that constructor injection is not suitable for working with optional dependencies. It is also more difficult to use in combination with class hierarchies: if a class uses constructor injection then extending it and overriding the constructor becomes problematic.

Setter Injection

Another possible injection point into a class is by adding a setter method that accepts the dependency:

Listing 24-3
```php
class NewsletterManager
{
    protected $mailer;

    public function setMailer(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Listing 24-4
```yaml
services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     NewsletterManager
        calls:
            - [setMailer, ["@my_mailer"]]
```

This time the advantages are:

- Setter injection works well with optional dependencies. If you do not need the dependency, then just do not call the setter.
- You can call the setter multiple times. This is particularly useful if the method adds the dependency to a collection. You can then have a variable number of dependencies.

The disadvantages of setter injection are:

- The setter can be called more than just at the time of construction so you cannot be sure the dependency is not replaced during the lifetime of the object (except by explicitly writing the setter method to check if it has already been called).

- You cannot be sure the setter will be called and so you need to add checks that any required dependencies are injected.

Property Injection

Another possibility is just setting public fields of the class directly:

Listing 24-5
```php
class NewsletterManager
{
    public $mailer;

    // ...
}
```

Listing 24-6
```yaml
services:
     my_mailer:
         # ...
     newsletter_manager:
         class: NewsletterManager
         properties:
             mailer: "@my_mailer"
```

There are mainly only disadvantages to using property injection, it is similar to setter injection but with these additional important problems:

- You cannot control when the dependency is set at all, it can be changed at any point in the object's lifetime.
- You cannot use type hinting so you cannot be sure what dependency is injected except by writing into the class code to explicitly test the class instance before using it.

But, it is useful to know that this can be done with the service container, especially if you are working with code that is out of your control, such as in a third party library, which uses public properties for its dependencies.

Chapter 25
Introduction to Parameters

You can define parameters in the service container which can then be used directly or as part of service definitions. This can help to separate out values that you will want to change more regularly.

Getting and Setting Container Parameters

Working with container parameters is straightforward using the container's accessor methods for parameters. You can check if a parameter has been defined in the container with:

Listing 25-1
```php
$container->hasParameter('mailer.transport');
```

You can retrieve a parameter set in the container with:

Listing 25-2
```php
$container->getParameter('mailer.transport');
```

and set a parameter in the container with:

Listing 25-3
```php
$container->setParameter('mailer.transport', 'sendmail');
```

The used `.` notation is just a *Symfony convention* to make parameters easier to read. Parameters are just flat key-value elements, they can't be organized into a nested array

You can only set a parameter before the container is compiled. To learn more about compiling the container see *Compiling the Container*.

Parameters in Configuration Files

You can also use the `parameters` section of a config file to set parameters:

Listing 25-4
```
parameters:
    mailer.transport: sendmail
```

As well as retrieving the parameter values directly from the container you can use them in the config files. You can refer to parameters elsewhere by surrounding them with percent (`%`) signs, e.g. `%mailer.transport%`. One use for this is to inject the values into your services. This allows you to configure different versions of services between applications or multiple services based on the same class but configured differently within a single application. You could inject the choice of mail transport into the `Mailer` class directly. But declaring it as a parameter makes it easier to change rather than being tied up and hidden with the service definition:

Listing 25-5
```
parameters:
    mailer.transport: sendmail

services:
    mailer:
        class:     Mailer
        arguments: ['%mailer.transport%']
```

The values between `parameter` tags in XML configuration files are not trimmed.

This means that the following configuration sample will have the value `\n sendmail\n`:

Listing 25-6
```
<parameter key="mailer.transport">
    sendmail
</parameter>
```

In some cases (for constants or class names), this could throw errors. In order to prevent this, you must always inline your parameters as follow:

Listing 25-7
```
<parameter key="mailer.transport">sendmail</parameter>
```

If you were using this elsewhere as well, then you would only need to change the parameter value in one place if needed.

The percent sign inside a parameter or argument, as part of the string, must be escaped with another percent sign:

Listing 25-8
```
arguments: ["http://symfony.com/?foo=%%s&bar=%%d"]
```

Array Parameters

Parameters do not need to be flat strings, they can also contain array values. For the XML format, you need to use the `type="collection"` attribute for all parameters that are arrays.

Listing 25-9

```yaml
parameters:
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3
    my_multilang.language_fallback:
        en:
            - en
            - fr
        fr:
            - fr
            - en
```

Constants as Parameters

The container also has support for setting PHP constants as parameters. To take advantage of this feature, map the name of your constant to a parameter key, and define the type as `constant`.

Listing 25-10
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/
dic/services/services-1.0.xsd">

    <parameters>
        <parameter key="global.constant.value" type="constant">GLOBAL_CONSTANT</parameter>
        <parameter key="my_class.constant.value"
type="constant">My_Class::CONSTANT_NAME</parameter>
    </parameters>
</container>
```

This does not work for YAML configurations. If you're using YAML, you can import an XML file to take advantage of this functionality:

Listing 25-11
```yaml
imports:
    - { resource: parameters.xml }
```

PHP Keywords in XML

By default, `true`, `false` and `null` in XML are converted to the PHP keywords (respectively `true`, `false` and `null`):

Listing 25-12
```xml
<parameters>
    <parameter key="mailer.send_all_in_once">false</parameter>
</parameters>

<!-- after parsing
$container->getParameter('mailer.send_all_in_once'); // returns false
-->
```

To disable this behavior, use the `string` type:

Listing 25-13
```xml
<parameters>
    <parameter key="mailer.some_parameter" type="string">true</parameter>
</parameters>

<!-- after parsing
$container->getParameter('mailer.some_parameter'); // returns "true"
-->
```

This is not available for YAML and PHP, because they already have built-in support for the PHP keywords.

Chapter 26

Working with Container Service Definitions

Getting and Setting Service Definitions

There are some helpful methods for working with the service definitions.

To find out if there is a definition for a service id:

Listing 26-1
```
$container->hasDefinition($serviceId);
```

This is useful if you only want to do something if a particular definition exists.

You can retrieve a definition with:

Listing 26-2
```
$container->getDefinition($serviceId);
```

or:

Listing 26-3
```
$container->findDefinition($serviceId);
```

which unlike `getDefinition()` also resolves aliases so if the `$serviceId` argument is an alias you will get the underlying definition.

The service definitions themselves are objects so if you retrieve a definition with these methods and make changes to it these will be reflected in the container. If, however, you are creating a new definition then you can add it to the container using:

Listing 26-4
```
$container->setDefinition($id, $definition);
```

Working with a Definition

Creating a new Definition

If you need to create a new definition rather than manipulate one retrieved from the container then the definition class is *`Definition`*[1].

Class

First up is the class of a definition, this is the class of the object returned when the service is requested from the container.

To find out what class is set for a definition:

Listing 26-5
```php
$definition->getClass();
```

and to set a different class:

Listing 26-6
```php
$definition->setClass($class); // Fully qualified class name as string
```

Constructor Arguments

To get an array of the constructor arguments for a definition you can use:

Listing 26-7
```php
$definition->getArguments();
```

or to get a single argument by its position:

Listing 26-8
```php
$definition->getArgument($index);
// e.g. $definition->getArgument(0) for the first argument
```

You can add a new argument to the end of the arguments array using:

Listing 26-9
```php
$definition->addArgument($argument);
```

The argument can be a string, an array, a service parameter by using `%parameter_name%` or a service id by using:

Listing 26-10
```php
use Symfony\Component\DependencyInjection\Reference;

// ...

$definition->addArgument(new Reference('service_id'));
```

In a similar way you can replace an already set argument by index using:

Listing 26-11
```php
$definition->replaceArgument($index, $argument);
```

You can also replace all the arguments (or set some if there are none) with an array of arguments:

Listing 26-12

1. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Definition.html

```php
$definition->setArguments($arguments);
```

Method Calls

If the service you are working with uses setter injection then you can manipulate any method calls in the definitions as well.

You can get an array of all the method calls with:

Listing 26-13
```php
$definition->getMethodCalls();
```

Add a method call with:

Listing 26-14
```php
$definition->addMethodCall($method, $arguments);
```

Where `$method` is the method name and `$arguments` is an array of the arguments to call the method with. The arguments can be strings, arrays, parameters or service ids as with the constructor arguments.

You can also replace any existing method calls with an array of new ones with:

Listing 26-15
```php
$definition->setMethodCalls($methodCalls);
```

There are more examples of specific ways of working with definitions in the PHP code blocks of the configuration examples on pages such as *Using a Factory to Create Services* and *Managing common Dependencies with parent Services*.

The methods here that change service definitions can only be used before the container is compiled. Once the container is compiled you cannot manipulate service definitions further. To learn more about compiling the container see *Compiling the Container*.

Chapter 27
Compiling the Container

The service container can be compiled for various reasons. These reasons include checking for any potential issues such as circular references and making the container more efficient by resolving parameters and removing unused services. Also, certain features - like using *parent services* - require the container to be compiled.

It is compiled by running:

Listing 27-1
```
$container->compile();
```

The compile method uses *Compiler Passes* for the compilation. The DependencyInjection component comes with several passes which are automatically registered for compilation. For example the *`CheckDefinitionValidityPass`*[1] checks for various potential issues with the definitions that have been set in the container. After this and several other passes that check the container's validity, further compiler passes are used to optimize the configuration before it is cached. For example, private services and abstract services are removed, and aliases are resolved.

Managing Configuration with Extensions

As well as loading configuration directly into the container as shown in *The DependencyInjection Component*, you can manage it by registering extensions with the container. The first step in the compilation process is to load configuration from any extension classes registered with the container. Unlike the configuration loaded directly, they are only processed when the container is compiled. If your application is modular then extensions allow each module to register and manage their own service configuration.

The extensions must implement *`ExtensionInterface`*[2] and can be registered with the container with:

Listing 27-2
```
$container->registerExtension($extension);
```

1. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Compiler/CheckDefinitionValidityPass.html
2. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html

The main work of the extension is done in the `load` method. In the `load` method you can load configuration from one or more configuration files as well as manipulate the container definitions using the methods shown in *Working with Container Service Definitions*.

The `load` method is passed a fresh container to set up, which is then merged afterwards into the container it is registered with. This allows you to have several extensions managing container definitions independently. The extensions do not add to the containers configuration when they are added but are processed when the container's `compile` method is called.

A very simple extension may just load configuration files into the container:

Listing 27-3
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\DependencyInjection\Extension\ExtensionInterface;
use Symfony\Component\Config\FileLocator;

class AcmeDemoExtension implements ExtensionInterface
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader(
            $container,
            new FileLocator(__DIR__.'/../Resources/config')
        );
        $loader->load('services.xml');
    }

    // ...
}
```

This does not gain very much compared to loading the file directly into the overall container being built. It just allows the files to be split up amongst the modules/bundles. Being able to affect the configuration of a module from configuration files outside of the module/bundle is needed to make a complex application configurable. This can be done by specifying sections of config files loaded directly into the container as being for a particular extension. These sections on the config will not be processed directly by the container but by the relevant Extension.

The Extension must specify a `getAlias` method to implement the interface:

Listing 27-4
```php
// ...

class AcmeDemoExtension implements ExtensionInterface
{
    // ...

    public function getAlias()
    {
        return 'acme_demo';
    }
}
```

For YAML configuration files specifying the alias for the Extension as a key will mean that those values are passed to the Extension's `load` method:

Listing 27-5
```yaml
# ...
acme_demo:
    foo: fooValue
    bar: barValue
```

If this file is loaded into the configuration then the values in it are only processed when the container is compiled at which point the Extensions are loaded:

Listing 27-6
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

$container = new ContainerBuilder();
$container->registerExtension(new AcmeDemoExtension);

$loader = new YamlFileLoader($container, new FileLocator(__DIR__));
$loader->load('config.yml');

// ...
$container->compile();
```

When loading a config file that uses an extension alias as a key, the extension must already have been registered with the container builder or an exception will be thrown.

The values from those sections of the config files are passed into the first argument of the `load` method of the extension:

Listing 27-7
```php
public function load(array $configs, ContainerBuilder $container)
{
    $foo = $configs[0]['foo']; //fooValue
    $bar = $configs[0]['bar']; //barValue
}
```

The `$configs` argument is an array containing each different config file that was loaded into the container. You are only loading a single config file in the above example but it will still be within an array. The array will look like this:

Listing 27-8
```php
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    ),
)
```

Whilst you can manually manage merging the different files, it is much better to use *the Config component* to merge and validate the config values. Using the configuration processing you could access the config value this way:

Listing 27-9
```php
use Symfony\Component\Config\Definition\Processor;
// ...

public function load(array $configs, ContainerBuilder $container)
{
    $configuration = new Configuration();
    $processor = new Processor();
    $config = $processor->processConfiguration($configuration, $configs);

    $foo = $config['foo']; //fooValue
    $bar = $config['bar']; //barValue
```

```
12
13      // ...
14  }
```

There are a further two methods you must implement. One to return the XML namespace so that the relevant parts of an XML config file are passed to the extension. The other to specify the base path to XSD files to validate the XML configuration:

Listing 27-10
```
1  public function getXsdValidationBasePath()
2  {
3      return __DIR__.'/../Resources/config/';
4  }
5
6  public function getNamespace()
7  {
8      return 'http://www.example.com/symfony/schema/';
9  }
```

XSD validation is optional, returning `false` from the `getXsdValidationBasePath` method will disable it.

The XML version of the config would then look like this:

Listing 27-11
```
1  <?xml version="1.0" ?>
2  <container xmlns="http://symfony.com/schema/dic/services"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:acme_demo="http://www.example.com/symfony/schema/"
5      xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/
6  symfony/schema/hello-1.0.xsd">
7
8      <acme_demo:config>
9          <acme_demo:foo>fooValue</acme_hello:foo>
10         <acme_demo:bar>barValue</acme_demo:bar>
11     </acme_demo:config>
   </container>
```

In the Symfony full stack framework there is a base Extension class which implements these methods as well as a shortcut method for processing the configuration. See *How to Load Service Configuration inside a Bundle* for more details.

The processed config value can now be added as container parameters as if it were listed in a `parameters` section of the config file but with the additional benefit of merging multiple files and validation of the configuration:

Listing 27-12
```
1  public function load(array $configs, ContainerBuilder $container)
2  {
3      $configuration = new Configuration();
4      $processor = new Processor();
5      $config = $processor->processConfiguration($configuration, $configs);
6
7      $container->setParameter('acme_demo.FOO', $config['foo']);
```

```
8
9       // ...
10  }
```

More complex configuration requirements can be catered for in the Extension classes. For example, you may choose to load a main service configuration file but also load a secondary one only if a certain parameter is set:

Listing 27-13
```
1  public function load(array $configs, ContainerBuilder $container)
2  {
3      $configuration = new Configuration();
4      $processor = new Processor();
5      $config = $processor->processConfiguration($configuration, $configs);
6
7      $loader = new XmlFileLoader(
8          $container,
9          new FileLocator(__DIR__.'/../Resources/config')
10     );
11     $loader->load('services.xml');
12
13     if ($config['advanced']) {
14         $loader->load('advanced.xml');
15     }
16 }
```

Just registering an extension with the container is not enough to get it included in the processed extensions when the container is compiled. Loading config which uses the extension's alias as a key as in the above examples will ensure it is loaded. The container builder can also be told to load it with its *loadFromExtension()*[3] method:

Listing 27-14
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2
3  $container = new ContainerBuilder();
4  $extension = new AcmeDemoExtension();
5  $container->registerExtension($extension);
6  $container->loadFromExtension($extension->getAlias());
7  $container->compile();
```

If you need to manipulate the configuration loaded by an extension then you cannot do it from another extension as it uses a fresh container. You should instead use a compiler pass which works with the full container after the extensions have been processed.

Prepending Configuration Passed to the Extension

New in version 2.2: The ability to prepend the configuration of a bundle was introduced in Symfony 2.2.

An Extension can prepend the configuration of any Bundle before the `load()` method is called by implementing *PrependExtensionInterface*[4]:

3. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/ContainerBuilder.html#loadFromExtension()
4. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Extension/PrependExtensionInterface.html

Listing 27-15
```php
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
// ...

class AcmeDemoExtension implements ExtensionInterface, PrependExtensionInterface
{
    // ...

    public function prepend()
    {
        // ...

        $container->prependExtensionConfig($name, $config);

        // ...
    }
}
```

For more details, see *How to Simplify Configuration of multiple Bundles*, which is specific to the Symfony Framework, but contains more details about this feature.

Creating a Compiler Pass

You can also create and register your own compiler passes with the container. To create a compiler pass it needs to implement the *`CompilerPassInterface`*[5] interface. The compiler pass gives you an opportunity to manipulate the service definitions that have been compiled. This can be very powerful, but is not something needed in everyday use.

The compiler pass must have the `process` method which is passed the container being compiled:

Listing 27-16
```php
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class CustomCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
       // ...
    }
}
```

The container's parameters and definitions can be manipulated using the methods described in the *Working with Container Service Definitions*. One common thing to do in a compiler pass is to search for all services that have a certain tag in order to process them in some way or dynamically plug each into some other service.

Registering a Compiler Pass

You need to register your custom pass with the container. Its process method will then be called when the container is compiled:

Listing 27-17

5. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Compiler/CompilerPassInterface.html

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->addCompilerPass(new CustomCompilerPass);
```

Compiler passes are registered differently if you are using the full stack framework, see *How to Work with Compiler Passes in Bundles* for more details.

Controlling the Pass Ordering

The default compiler passes are grouped into optimization passes and removal passes. The optimization passes run first and include tasks such as resolving references within the definitions. The removal passes perform tasks such as removing private aliases and unused services. You can choose where in the order any custom passes you add are run. By default they will be run before the optimization passes.

You can use the following constants as the second argument when registering a pass with the container to control where it goes in the order:

- `PassConfig::TYPE_BEFORE_OPTIMIZATION`
- `PassConfig::TYPE_OPTIMIZE`
- `PassConfig::TYPE_BEFORE_REMOVING`
- `PassConfig::TYPE_REMOVE`
- `PassConfig::TYPE_AFTER_REMOVING`

For example, to run your custom pass after the default removal passes have been run:

Listing 27-18
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\PassConfig;

$container = new ContainerBuilder();
$container->addCompilerPass(
    new CustomCompilerPass,
    PassConfig::TYPE_AFTER_REMOVING
);
```

Dumping the Configuration for Performance

Using configuration files to manage the service container can be much easier to understand than using PHP once there are a lot of services. This ease comes at a price though when it comes to performance as the config files need to be parsed and the PHP configuration built from them. The compilation process makes the container more efficient but it takes time to run. You can have the best of both worlds though by using configuration files and then dumping and caching the resulting configuration. The `PhpDumper` makes dumping the compiled container easy:

Listing 27-19
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Dumper\PhpDumper;

$file = __DIR__ .'/cache/container.php';

if (file_exists($file)) {
```

```
    require_once $file;
    $container = new ProjectServiceContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    $dumper = new PhpDumper($container);
    file_put_contents($file, $dumper->dump());
}
```

`ProjectServiceContainer` is the default name given to the dumped container class, you can change this though this with the `class` option when you dump it:

Listing 27-20
```
// ...
$file = __DIR__ .'/cache/container.php';

if (file_exists($file)) {
    require_once $file;
    $container = new MyCachedContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    $dumper = new PhpDumper($container);
    file_put_contents(
        $file,
        $dumper->dump(array('class' => 'MyCachedContainer'))
    );
}
```

You will now get the speed of the PHP configured container with the ease of using configuration files. Additionally dumping the container in this way further optimizes how the services are created by the container.

In the above example you will need to delete the cached container file whenever you make any changes. Adding a check for a variable that determines if you are in debug mode allows you to keep the speed of the cached container in production but getting an up to date configuration whilst developing your application:

Listing 27-21
```
// ...

// based on something in your project
$isDebug = ...;

$file = __DIR__ .'/cache/container.php';

if (!$isDebug && file_exists($file)) {
    require_once $file;
    $container = new MyCachedContainer();
} else {
    $container = new ContainerBuilder();
    // ...
    $container->compile();

    if (!$isDebug) {
```

```
        $dumper = new PhpDumper($container);
        file_put_contents(
            $file,
            $dumper->dump(array('class' => 'MyCachedContainer'))
        );
    }
}
```

This could be further improved by only recompiling the container in debug mode when changes have been made to its configuration rather than on every request. This can be done by caching the resource files used to configure the container in the way described in "*Caching Based on Resources*" in the config component documentation.

You do not need to work out which files to cache as the container builder keeps track of all the resources used to configure it, not just the configuration files but the extension classes and compiler passes as well. This means that any changes to any of these files will invalidate the cache and trigger the container being rebuilt. You just need to ask the container for these resources and use them as metadata for the cache:

Listing 27-22
```
// ...

// based on something in your project
$isDebug = ...;

$file = __DIR__ .'/cache/container.php';
$containerConfigCache = new ConfigCache($file, $isDebug);

if (!$containerConfigCache->isFresh()) {
    $containerBuilder = new ContainerBuilder();
    // ...
    $containerBuilder->compile();

    $dumper = new PhpDumper($containerBuilder);
    $containerConfigCache->write(
        $dumper->dump(array('class' => 'MyCachedContainer')),
        $containerBuilder->getResources()
    );
}

require_once $file;
$container = new MyCachedContainer();
```

Now the cached dumped container is used regardless of whether debug mode is on or not. The difference is that the `ConfigCache` is set to debug mode with its second constructor argument. When the cache is not in debug mode the cached container will always be used if it exists. In debug mode, an additional metadata file is written with the timestamps of all the resource files. These are then checked to see if the files have changed, if they have the cache will be considered stale.

In the full stack framework the compilation and caching of the container is taken care of for you.

Chapter 28
Working with Tagged Services

Tags are a generic string (along with some options) that can be applied to any service. By themselves, tags don't actually alter the functionality of your services in any way. But if you choose to, you can ask a container builder for a list of all services that were tagged with some specific tag. This is useful in compiler passes where you can find these services and use or modify them in some specific way.

For example, if you are using Swift Mailer you might imagine that you want to implement a "transport chain", which is a collection of classes implementing `\Swift_Transport`. Using the chain, you'll want Swift Mailer to try several ways of transporting the message until one succeeds.

To begin with, define the `TransportChain` class:

Listing 28-1

```
class TransportChain
{
    private $transports;

    public function __construct()
    {
        $this->transports = array();
    }

    public function addTransport(\Swift_Transport $transport)
    {
        $this->transports[] = $transport;
    }
}
```

Then, define the chain as a service:

Listing 28-2

```
services:
    acme_mailer.transport_chain:
        class: TransportChain
```

Define Services with a custom Tag

Now you might want several of the `\Swift_Transport` classes to be instantiated and added to the chain automatically using the `addTransport()` method. For example you may add the following transports as services:

Listing 28-3
```yaml
services:
    acme_mailer.transport.smtp:
        class: \Swift_SmtpTransport
        arguments:
            - "%mailer_host%"
        tags:
            -  { name: acme_mailer.transport }
    acme_mailer.transport.sendmail:
        class: \Swift_SendmailTransport
        tags:
            -  { name: acme_mailer.transport }
```

Notice that each was given a tag named `acme_mailer.transport`. This is the custom tag that you'll use in your compiler pass. The compiler pass is what makes this tag "mean" something.

Create a `CompilerPass`

Your compiler pass can now ask the container for any services with the custom tag:

Listing 28-4
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (!$container->has('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->findDefinition(
            'acme_mailer.transport_chain'
        );

        $taggedServices = $container->findTaggedServiceIds(
            'acme_mailer.transport'
        );
        foreach ($taggedServices as $id => $tags) {
            $definition->addMethodCall(
                'addTransport',
                array(new Reference($id))
            );
        }
    }
}
```

The `process()` method checks for the existence of the `acme_mailer.transport_chain` service, then looks for all services tagged `acme_mailer.transport`. It adds to the definition of the

`acme_mailer.transport_chain` service a call to `addTransport()` for each "acme_mailer.transport" service it has found. The first argument of each of these calls will be the mailer transport service itself.

Register the Pass with the Container

You also need to register the pass with the container, it will then be run when the container is compiled:

Listing 28-5
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->addCompilerPass(new TransportCompilerPass());
```

Compiler passes are registered differently if you are using the full stack framework. See *How to Work with Compiler Passes in Bundles* for more details.

Adding additional Attributes on Tags

Sometimes you need additional information about each service that's tagged with your tag. For example, you might want to add an alias to each member of the transport chain.

To begin with, change the `TransportChain` class:

Listing 28-6
```php
class TransportChain
{
    private $transports;

    public function __construct()
    {
        $this->transports = array();
    }

    public function addTransport(\Swift_Transport $transport, $alias)
    {
        $this->transports[$alias] = $transport;
    }

    public function getTransport($alias)
    {
        if (array_key_exists($alias, $this->transports)) {
            return $this->transports[$alias];
        }
    }
}
```

As you can see, when `addTransport` is called, it takes not only a `Swift_Transport` object, but also a string alias for that transport. So, how can you allow each tagged transport service to also supply an alias?

To answer this, change the service declaration:

Listing 28-7
```yaml
services:
    acme_mailer.transport.smtp:
        class: \Swift_SmtpTransport
```

```
        arguments:
            - "%mailer_host%"
        tags:
            -  { name: acme_mailer.transport, alias: foo }
    acme_mailer.transport.sendmail:
        class: \Swift_SendmailTransport
        tags:
            -  { name: acme_mailer.transport, alias: bar }
```

Notice that you've added a generic `alias` key to the tag. To actually use this, update the compiler:

Listing 28-8

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (!$container->hasDefinition('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->getDefinition(
            'acme_mailer.transport_chain'
        );

        $taggedServices = $container->findTaggedServiceIds(
            'acme_mailer.transport'
        );
        foreach ($taggedServices as $id => $tags) {
            foreach ($tags as $attributes) {
                $definition->addMethodCall(
                    'addTransport',
                    array(new Reference($id), $attributes["alias"])
                );
            }
        }
    }
}
```

The double loop may be confusing. This is because a service can have more than one tag. You tag a service twice or more with the `acme_mailer.transport` tag. The second foreach loop iterates over the `acme_mailer.transport` tags set for the current service and gives you the attributes.

Chapter 29
Using a Factory to Create Services

Symfony's Service Container provides a powerful way of controlling the creation of objects, allowing you to specify arguments passed to the constructor as well as calling methods and setting parameters. Sometimes, however, this will not provide you with everything you need to construct your objects. For this situation, you can use a factory to create the object and tell the service container to call a method on the factory rather than directly instantiating the class.

Suppose you have a factory that configures and returns a new `NewsletterManager` object:

Listing 29-1
```php
class NewsletterManagerFactory
{
    public static function createNewsletterManager()
    {
        $newsletterManager = new NewsletterManager();

        // ...

        return $newsletterManager;
    }
}
```

To make the `NewsletterManager` object available as a service, you can configure the service container to use the `NewsletterManagerFactory` factory class:

Listing 29-2
```yaml
services:
    newsletter_manager:
        class:          NewsletterManager
        factory_class:  NewsletterManagerFactory
        factory_method: createNewsletterManager
```

When using a factory to create services, the value chosen for the `class` option has no effect on the resulting service. The actual class name only depends on the object that is returned by the factory. However, the configured class name may be used by compiler passes and therefore should be set to a sensible value.

When you specify the class to use for the factory (via `factory_class`) the method will be called statically. If the factory itself should be instantiated and the resulting object's method called, configure the factory itself as a service. In this case, the method (e.g. `createNewsletterManager`) should be changed to be non-static:

Listing 29-3
```yaml
services:
    newsletter_manager_factory:
        class:            NewsletterManagerFactory
    newsletter_manager:
        class:            NewsletterManager
        factory_service:  newsletter_manager_factory
        factory_method:   createNewsletterManager
```

The factory service is specified by its id name and not a reference to the service itself. So, you do not need to use the @ syntax for this in YAML configurations.

Passing Arguments to the Factory Method

If you need to pass arguments to the factory method, you can use the `arguments` options inside the service container. For example, suppose the `createNewsletterManager` method in the previous example takes the `templating` service as an argument:

Listing 29-4
```yaml
services:
    newsletter_manager_factory:
        class:            NewsletterManagerFactory
    newsletter_manager:
        class:            NewsletterManager
        factory_service:  newsletter_manager_factory
        factory_method:   createNewsletterManager
        arguments:
            - "@templating"
```

Chapter 30
Configuring Services with a Service Configurator

The Service Configurator is a feature of the Dependency Injection Container that allows you to use a callable to configure a service after its instantiation.

You can specify a method in another service, a PHP function or a static method in a class. The service instance is passed to the callable, allowing the configurator to do whatever it needs to configure the service after its creation.

A Service Configurator can be used, for example, when you have a service that requires complex setup based on configuration settings coming from different sources/services. Using an external configurator, you can maintain the service implementation cleanly and keep it decoupled from the other objects that provide the configuration needed.

Another interesting use case is when you have multiple objects that share a common configuration or that should be configured in a similar way at runtime.

For example, suppose you have an application where you send different types of emails to users. Emails are passed through different formatters that could be enabled or not depending on some dynamic application settings. You start defining a `NewsletterManager` class like this:

Listing 30-1

```
class NewsletterManager implements EmailFormatterAwareInterface
{
    protected $mailer;
    protected $enabledFormatters;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEnabledFormatters(array $enabledFormatters)
    {
        $this->enabledFormatters = $enabledFormatters;
    }

```

```
16      // ...
17  }
```

and also a `GreetingCardManager` class:

Listing 30-2
```
1   class GreetingCardManager implements EmailFormatterAwareInterface
2   {
3       protected $mailer;
4       protected $enabledFormatters;
5
6       public function setMailer(Mailer $mailer)
7       {
8           $this->mailer = $mailer;
9       }
10
11      public function setEnabledFormatters(array $enabledFormatters)
12      {
13          $this->enabledFormatters = $enabledFormatters;
14      }
15
16      // ...
17  }
```

As mentioned before, the goal is to set the formatters at runtime depending on application settings. To do this, you also have an `EmailFormatterManager` class which is responsible for loading and validating formatters enabled in the application:

Listing 30-3
```
1   class EmailFormatterManager
2   {
3       protected $enabledFormatters;
4
5       public function loadFormatters()
6       {
7           // code to configure which formatters to use
8           $enabledFormatters = array(...);
9           // ...
10
11          $this->enabledFormatters = $enabledFormatters;
12      }
13
14      public function getEnabledFormatters()
15      {
16          return $this->enabledFormatters;
17      }
18
19      // ...
20  }
```

If your goal is to avoid having to couple `NewsletterManager` and `GreetingCardManager` with `EmailFormatterManager`, then you might want to create a configurator class to configure these instances:

Listing 30-4
```
1   class EmailConfigurator
2   {
3       private $formatterManager;
4
5       public function __construct(EmailFormatterManager $formatterManager)
```

```
    {
        $this->formatterManager = $formatterManager;
    }

    public function configure(EmailFormatterAwareInterface $emailManager)
    {
        $emailManager->setEnabledFormatters(
            $this->formatterManager->getEnabledFormatters()
        );
    }

    // ...
}
```

The `EmailConfigurator`'s job is to inject the enabled filters into `NewsletterManager` and `GreetingCardManager` because they are not aware of where the enabled filters come from. In the other hand, the `EmailFormatterManager` holds the knowledge about the enabled formatters and how to load them, keeping the single responsibility principle.

Configurator Service Config

The service config for the above classes would look something like this:

Listing 30-5

```
services:
    my_mailer:
        # ...

    email_formatter_manager:
        class:     EmailFormatterManager
        # ...

    email_configurator:
        class:     EmailConfigurator
        arguments: ["@email_formatter_manager"]
        # ...

    newsletter_manager:
        class:     NewsletterManager
        calls:
            - [setMailer, ["@my_mailer"]]
        configurator: ["@email_configurator", configure]

    greeting_card_manager:
        class:     GreetingCardManager
        calls:
            - [setMailer, ["@my_mailer"]]
        configurator: ["@email_configurator", configure]
```

Chapter 31

Managing common Dependencies with parent Services

As you add more functionality to your application, you may well start to have related classes that share some of the same dependencies. For example you may have a Newsletter Manager which uses setter injection to set its dependencies:

Listing 31-1
```php
class NewsletterManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }

    // ...
}
```

and also a Greeting Card class which shares the same dependencies:

Listing 31-2
```php
class GreetingCardManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
```

```
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }

    // ...
}
```

The service config for these classes would look something like this:

Listing 31-3

```
services:
    my_mailer:
        # ...

    my_email_formatter:
        # ...

    newsletter_manager:
        class: NewsletterManager
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    greeting_card_manager:
        class: "GreetingCardManager"
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]
```

There is a lot of repetition in both the classes and the configuration. This means that if you changed, for example, the `Mailer` of `EmailFormatter` classes to be injected via the constructor, you would need to update the config in two places. Likewise if you needed to make changes to the setter methods you would need to do this in both classes. The typical way to deal with the common methods of these related classes would be to extract them to a super class:

Listing 31-4

```
abstract class MailManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }

    // ...
}
```

The `NewsletterManager` and `GreetingCardManager` can then extend this super class:

Listing 31-5

```
class NewsletterManager extends MailManager
{
    // ...
}
```

and:

Listing 31-6
```
class GreetingCardManager extends MailManager
{
    // ...
}
```

In a similar fashion, the Symfony service container also supports extending services in the configuration so you can also reduce the repetition by specifying a parent for a service.

Listing 31-7
```
# ...
services:
    # ...
    mail_manager:
        abstract:  true
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    newsletter_manager:
        class:  "NewsletterManager"
        parent: mail_manager

    greeting_card_manager:
        class:  "GreetingCardManager"
        parent: mail_manager
```

In this context, having a `parent` service implies that the arguments and method calls of the parent service should be used for the child services. Specifically, the setter methods defined for the parent service will be called when the child services are instantiated.

If you remove the `parent` config key, the services will still be instantiated and they will still of course extend the `MailManager` class. The difference is that omitting the `parent` config key will mean that the `calls` defined on the `mail_manager` service will not be executed when the child services are instantiated.

The `scope`, `abstract` and `tags` attributes are always taken from the child service.

The parent service is abstract as it should not be directly retrieved from the container or passed into another service. It exists merely as a "template" that other services can use. This is why it can have no `class` configured which would cause an exception to be raised for a non-abstract service.

In order for parent dependencies to resolve, the `ContainerBuilder` must first be compiled. See *Compiling the Container* for more details.

In the examples shown, the classes sharing the same configuration also extend from the same parent class in PHP. This isn't necessary at all. You can just extract common parts of similar service definitions into a parent service without also extending a parent class in PHP.

Overriding parent Dependencies

There may be times where you want to override what class is passed in for a dependency of one child service only. Fortunately, by adding the method call config for the child service, the dependencies set by the parent class will be overridden. So if you needed to pass a different dependency just to the `NewsletterManager` class, the config would look like this:

Listing 31-8

```yaml
# ...
services:
    # ...
    my_alternative_mailer:
        # ...

    mail_manager:
        abstract: true
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    newsletter_manager:
        class:  "NewsletterManager"
        parent: mail_manager
        calls:
            - [setMailer, ["@my_alternative_mailer"]]

    greeting_card_manager:
        class:  "GreetingCardManager"
        parent: mail_manager
```

The `GreetingCardManager` will receive the same dependencies as before, but the `NewsletterManager` will be passed the `my_alternative_mailer` instead of the `my_mailer` service.

You can't override method calls. When you defined new method calls in the child service, it'll be added to the current set of configured method calls. This means it works perfectly when the setter overrides the current property, but it doesn't work as expected when the setter appends it to the existing data (e.g. an `addFilters()` method). In those cases, the only solution is to *not* extend the parent service and configuring the service just like you did before knowing this feature.

Chapter 32
Advanced Container Configuration

Marking Services as public / private

When defining services, you'll usually want to be able to access these definitions within your application code. These services are called `public`. For example, the `doctrine` service registered with the container when using the DoctrineBundle is a public service. This means that you can fetch it from the container using the `get()` method:

Listing 32-1
```
$doctrine = $container->get('doctrine');
```

In some cases, a service *only* exists to be injected into another service and is *not* intended to be fetched directly from the container as shown above.

In these cases, to get a minor performance boost, you can set the service to be *not* public (i.e. private):

Listing 32-2
```
services:
    foo:
      class: Example\Foo
      public: false
```

What makes private services special is that, if they are only injected once, they are converted from services to inlined instantiations (e.g. `new PrivateThing()`). This increases the container's performance.

Now that the service is private, you *should not* fetch the service directly from the container:

Listing 32-3
```
$container->get('foo');
```

This *may or may not work*, depending on if the service could be inlined. Simply said: A service can be marked as private if you do not want to access it directly from your code.

However, if a service has been marked as private, you can still alias it (see below) to access this service (via the alias).

Services are by default public.

Synthetic Services

Synthetic services are services that are injected into the container instead of being created by the container.

For example, if you're using the *HttpKernel* component with the DependencyInjection component, then the `request` service is injected in the *`ContainerAwareHttpKernel::handle()`*[1] method when entering the request *scope*. The class does not exist when there is no request, so it can't be included in the container configuration. Also, the service should be different for every subrequest in the application.

To create a synthetic service, set `synthetic` to `true`:

Listing 32-4
```yaml
services:
    request:
        synthetic: true
```

As you see, only the `synthetic` option is set. All other options are only used to configure how a service is created by the container. As the service isn't created by the container, these options are omitted.

Now, you can inject the class by using *`Container::set`*[2]:

Listing 32-5
```php
// ...
$container->set('request', new MyRequest(...));
```

Aliasing

You may sometimes want to use shortcuts to access some services. You can do so by aliasing them and, furthermore, you can even alias non-public services.

Listing 32-6
```yaml
services:
   foo:
     class: Example\Foo
   bar:
     alias: foo
```

This means that when using the container directly, you can access the `foo` service by asking for the `bar` service like this:

Listing 32-7
```php
$container->get('bar'); // Would return the foo service
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/DependencyInjection/ContainerAwareHttpKernel.html#handle()
2. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/Container.html#set()

In YAML, you can also use a shortcut to alias a service:

Listing 32-8
```yaml
services:
    foo:
      class: Example\Foo
    bar: "@foo"
```

Requiring Files

There might be use cases when you need to include another file just before the service itself gets loaded. To do so, you can use the `file` directive.

Listing 32-9
```yaml
services:
    foo:
      class: Example\Foo\Bar
      file: "%kernel.root_dir%/src/path/to/file/foo.php"
```

Notice that Symfony will internally call the PHP statement `require_once`, which means that your file will be included only once per request.

Chapter 33
Lazy Services

New in version 2.3: Lazy services were introduced in Symfony 2.3.

Why lazy Services?

In some cases, you may want to inject a service that is a bit heavy to instantiate, but is not always used inside your object. For example, imagine you have a `NewsletterManager` and you inject a `mailer` service into it. Only a few methods on your `NewsletterManager` actually use the `mailer`, but even when you don't need it, a `mailer` service is always instantiated in order to construct your `NewsletterManager`.

Configuring lazy services is one answer to this. With a lazy service, a "proxy" of the `mailer` service is actually injected. It looks and acts just like the `mailer`, except that the `mailer` isn't actually instantiated until you interact with the proxy in some way.

Installation

In order to use the lazy service instantiation, you will first need to install the *ProxyManager bridge*[1]:

Listing 33-1
```
$ composer require symfony/proxy-manager-bridge:~2.3
```

If you're using the full-stack framework, the proxy manager bridge is already included but the actual proxy manager needs to be included. So, run:

Listing 33-2
```
$ php composer.phar require ocramius/proxy-manager:~0.5
```

Afterwards compile your container and check to make sure that you get a proxy for your lazy services.

1. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/ProxyManager

Configuration

You can mark the service as `lazy` by manipulating its definition:

Listing 33-3
```yaml
services:
    foo:
      class: Acme\Foo
      lazy: true
```

You can then require the service from the container:

Listing 33-4
```php
$service = $container->get('foo');
```

At this point the retrieved `$service` should be a virtual *proxy*[2] with the same signature of the class representing the service. You can also inject the service just like normal into other services. The object that's actually injected will be the proxy.

To check if your proxy works you can simply check the interface of the received object.

Listing 33-5
```php
var_dump(class_implements($service));
```

If the class implements the `ProxyManager\Proxy\LazyLoadingInterface` your lazy loaded services are working.

If you don't install the *ProxyManager bridge*[3], the container will just skip over the `lazy` flag and simply instantiate the service as it would normally do.

The proxy gets initialized and the actual service is instantiated as soon as you interact in any way with this object.

Additional Resources

You can read more about how proxies are instantiated, generated and initialized in the *documentation of ProxyManager*[4].

2. http://en.wikipedia.org/wiki/Proxy_pattern
3. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/ProxyManager
4. https://github.com/Ocramius/ProxyManager/blob/master/docs/lazy-loading-value-holder.md

Chapter 34

Container Building Workflow

In the preceding pages of this section, there has been little to say about where the various files and classes should be located. This is because this depends on the application, library or framework in which you want to use the container. Looking at how the container is configured and built in the Symfony full stack framework will help you see how this all fits together, whether you are using the full stack framework or looking to use the service container in another application.

The full stack framework uses the HttpKernel component to manage the loading of the service container configuration from the application and bundles and also handles the compilation and caching. Even if you are not using HttpKernel, it should give you an idea of one way of organizing configuration in a modular application.

Working with a Cached Container

Before building it, the kernel checks to see if a cached version of the container exists. The HttpKernel has a debug setting and if this is false, the cached version is used if it exists. If debug is true then the kernel *checks to see if configuration is fresh* and if it is, the cached version of the container is used. If not then the container is built from the application-level configuration and the bundles's extension configuration.

Read *Dumping the Configuration for Performance* for more details.

Application-level Configuration

Application level config is loaded from the `app/config` directory. Multiple files are loaded which are then merged when the extensions are processed. This allows for different configuration for different environments e.g. dev, prod.

These files contain parameters and services that are loaded directly into the container as per *Setting Up the Container with Configuration Files*. They also contain configuration that is processed by extensions as per *Managing Configuration with Extensions*. These are considered to be bundle configuration since each bundle contains an Extension class.

Bundle-level Configuration with Extensions

By convention, each bundle contains an Extension class which is in the bundle's `DependencyInjection` directory. These are registered with the `ContainerBuilder` when the kernel is booted. When the `ContainerBuilder` is *compiled*, the application-level configuration relevant to the bundle's extension is passed to the Extension which also usually loads its own config file(s), typically from the bundle's `Resources/config` directory. The application-level config is usually processed with a *Configuration object* also stored in the bundle's `DependencyInjection` directory.

Compiler Passes to Allow Interaction between Bundles

Compiler passes are used to allow interaction between different bundles as they cannot affect each other's configuration in the extension classes. One of the main uses is to process tagged services, allowing bundles to register services to be picked up by other bundles, such as Monolog loggers, Twig extensions and Data Collectors for the Web Profiler. Compiler passes are usually placed in the bundle's `DependencyInjection/Compiler` directory.

Compilation and Caching

After the compilation process has loaded the services from the configuration, extensions and the compiler passes, it is dumped so that the cache can be used next time. The dumped version is then used during subsequent requests as it is more efficient.

Chapter 35
The DomCrawler Component

> The DomCrawler component eases DOM navigation for HTML and XML documents.

While possible, the DomCrawler component is not designed for manipulation of the DOM or re-dumping HTML/XML.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/dom-crawler` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/DomCrawler*[2]).

Usage

The *`Crawler`*[3] class provides methods to query and manipulate HTML and XML documents.

An instance of the Crawler represents a set (*`SplObjectStorage`*[4]) of *`DOMElement`*[5] objects, which are basically nodes that you can traverse easily:

Listing 35-1

```
1  use Symfony\Component\DomCrawler\Crawler;
2
3  $html = <<<'HTML'
```

1. https://packagist.org/packages/symfony/dom-crawler
2. https://github.com/symfony/DomCrawler
3. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html
4. http://php.net/manual/en/class.splobjectstorage.php
5. http://php.net/manual/en/class.domelement.php

```
<!DOCTYPE html>
<html>
    <body>
        <p class="message">Hello World!</p>
        <p>Hello Crawler!</p>
    </body>
</html>
HTML;

$crawler = new Crawler($html);

foreach ($crawler as $domElement) {
    print $domElement->nodeName;
}
```

Specialized *Link*[6] and *Form*[7] classes are useful for interacting with html links and forms as you traverse through the HTML tree.

The DomCrawler will attempt to automatically fix your HTML to match the official specification. For example, if you nest a `<p>` tag inside another `<p>` tag, it will be moved to be a sibling of the parent tag. This is expected and is part of the HTML5 spec. But if you're getting unexpected behavior, this could be a cause. And while the DomCrawler isn't meant to dump content, you can see the "fixed" version of your HTML by *dumping it*.

Node Filtering

Using XPath expressions is really easy:

Listing 35-2
```
$crawler = $crawler->filterXPath('descendant-or-self::body/p');
```

`DOMXPath::query` is used internally to actually perform an XPath query.

Filtering is even easier if you have the CssSelector component installed. This allows you to use jQuery-like selectors to traverse:

Listing 35-3
```
$crawler = $crawler->filter('body > p');
```

Anonymous function can be used to filter with more complex criteria:

Listing 35-4
```
use Symfony\Component\DomCrawler\Crawler;
// ...

$crawler = $crawler
    ->filter('body > p')
    ->reduce(function (Crawler $node, $i) {
        // filter even nodes
```

6. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Link.html
7. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Form.html

```
8           return ($i % 2) == 0;
9       });
```

To remove a node the anonymous function must return false.

All filter methods return a new *Crawler*[8] instance with filtered content.

Node Traversing

Access node by its position on the list:

Listing 35-5
```
1 $crawler->filter('body > p')->eq(0);
```

Get the first or last node of the current selection:

Listing 35-6
```
1 $crawler->filter('body > p')->first();
2 $crawler->filter('body > p')->last();
```

Get the nodes of the same level as the current selection:

Listing 35-7
```
1 $crawler->filter('body > p')->siblings();
```

Get the same level nodes after or before the current selection:

Listing 35-8
```
1 $crawler->filter('body > p')->nextAll();
2 $crawler->filter('body > p')->previousAll();
```

Get all the child or parent nodes:

Listing 35-9
```
1 $crawler->filter('body')->children();
2 $crawler->filter('body > p')->parents();
```

All the traversal methods return a new *Crawler*[9] instance.

Accessing Node Values

Access the value of the first node of the current selection:

Listing 35-10
```
1 $message = $crawler->filterXPath('//body/p')->text();
```

Access the attribute value of the first node of the current selection:

Listing 35-11

8. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html
9. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html

```php
$class = $crawler->filterXPath('//body/p')->attr('class');
```

Extract attribute and/or node values from the list of nodes:

Listing 35-12
```php
$attributes = $crawler
    ->filterXpath('//body/p')
    ->extract(array('_text', 'class'))
;
```

Special attribute `_text` represents a node value.

Call an anonymous function on each node of the list:

Listing 35-13
```php
use Symfony\Component\DomCrawler\Crawler;
// ...

$nodeValues = $crawler->filter('p')->each(function (Crawler $node, $i) {
    return $node->text();
});
```

New in version 2.3: As seen here, in Symfony 2.3, the `each` and `reduce` Closure functions are passed a `Crawler` as the first argument. Previously, that argument was a *DOMNode*[10].

The anonymous function receives the node (as a Crawler) and the position as arguments. The result is an array of values returned by the anonymous function calls.

Adding the Content

The crawler supports multiple ways of adding the content:

Listing 35-14
```php
$crawler = new Crawler('<html><body /></html>');

$crawler->addHtmlContent('<html><body /></html>');
$crawler->addXmlContent('<root><node /></root>');

$crawler->addContent('<html><body /></html>');
$crawler->addContent('<root><node /></root>', 'text/xml');

$crawler->add('<html><body /></html>');
$crawler->add('<root><node /></root>');
```

When dealing with character sets other than ISO-8859-1, always add HTML content using the *addHTMLContent()*[11] method where you can specify the second parameter to be your target character set.

As the Crawler's implementation is based on the DOM extension, it is also able to interact with native *DOMDocument*[12], *DOMNodeList*[13] and *DOMNode*[14] objects:

10. http://php.net/manual/en/class.domnode.php
11. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html#addHTMLContent()
12. http://php.net/manual/en/class.domdocument.php

Listing 35-15
```php
$document = new \DOMDocument();
$document->loadXml('<root><node /><node /></root>');
$nodeList = $document->getElementsByTagName('node');
$node = $document->getElementsByTagName('node')->item(0);

$crawler->addDocument($document);
$crawler->addNodeList($nodeList);
$crawler->addNodes(array($node));
$crawler->addNode($node);
$crawler->add($document);
```

Manipulating and Dumping a `Crawler`

These methods on the `Crawler` are intended to initially populate your `Crawler` and aren't intended to be used to further manipulate a DOM (though this is possible). However, since the `Crawler` is a set of *`DOMElement`*[15] objects, you can use any method or property available on *`DOMElement`*[16], *`DOMNode`*[17] or *`DOMDocument`*[18]. For example, you could get the HTML of a `Crawler` with something like this:

Listing 35-16
```php
$html = '';

foreach ($crawler as $domElement) {
    $html .= $domElement->ownerDocument->saveHTML($domElement);
}
```

Or you can get the HTML of the first node using *`html()`*[19]:

Listing 35-17
```php
$html = $crawler->html();
```

The `html` method is new in Symfony 2.3.

Links

To find a link by name (or a clickable image by its `alt` attribute), use the `selectLink` method on an existing crawler. This returns a Crawler instance with just the selected link(s). Calling `link()` gives you a special *`Link`*[20] object:

Listing 35-18
```php
$linksCrawler = $crawler->selectLink('Go elsewhere...');
$link = $linksCrawler->link();

// or do this all at once
$link = $crawler->selectLink('Go elsewhere...')->link();
```

The *`Link`*[21] object has several useful methods to get more information about the selected link itself:

13. http://php.net/manual/en/class.domnodelist.php
14. http://php.net/manual/en/class.domnode.php
15. http://php.net/manual/en/class.domelement.php
16. http://php.net/manual/en/class.domelement.php
17. http://php.net/manual/en/class.domnode.php
18. http://php.net/manual/en/class.domdocument.php
19. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Crawler.html#html()
20. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Link.html
21. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Link.html

Listing 35-19
```php
// return the proper URI that can be used to make another request
$uri = $link->getUri();
```

The `getUri()` is especially useful as it cleans the `href` value and transforms it into how it should really be processed. For example, for a link with `href="#foo"`, this would return the full URI of the current page suffixed with `#foo`. The return from `getUri()` is always a full URI that you can act on.

Forms

Special treatment is also given to forms. A `selectButton()` method is available on the Crawler which returns another Crawler that matches a button (`input[type=submit]`, `input[type=image]`, or a `button`) with the given text. This method is especially useful because you can use it to return a *Form*[22] object that represents the form that the button lives in:

Listing 35-20
```php
$form = $crawler->selectButton('validate')->form();

// or "fill" the form fields with data
$form = $crawler->selectButton('validate')->form(array(
    'name' => 'Ryan',
));
```

The *Form*[23] object has lots of very useful methods for working with forms:

Listing 35-21
```php
$uri = $form->getUri();

$method = $form->getMethod();
```

The *getUri()*[24] method does more than just return the `action` attribute of the form. If the form method is GET, then it mimics the browser's behavior and returns the `action` attribute followed by a query string of all of the form's values.

You can virtually set and get values on the form:

Listing 35-22
```php
// set values on the form internally
$form->setValues(array(
    'registration[username]' => 'symfonyfan',
    'registration[terms]'    => 1,
));

// get back an array of values - in the "flat" array like above
$values = $form->getValues();

// returns the values like PHP would see them,
// where "registration" is its own array
$values = $form->getPhpValues();
```

To work with multi-dimensional fields:

Listing 35-23

22. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Form.html
23. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Form.html
24. http://api.symfony.com/2.3/Symfony/Component/DomCrawler/Form.html#getUri()

```
<form>
    <input name="multi[]" />
    <input name="multi[]" />
    <input name="multi[dimensional]" />
</form>
```

Pass an array of values:

Listing 35-24
```
// Set a single field
$form->setValues(array('multi' => array('value')));

// Set multiple fields at once
$form->setValues(array('multi' => array(
    1             => 'value',
    'dimensional' => 'an other value'
)));
```

This is great, but it gets better! The `Form` object allows you to interact with your form like a browser, selecting radio values, ticking checkboxes, and uploading files:

Listing 35-25
```
$form['registration[username]']->setValue('symfonyfan');

// check or uncheck a checkbox
$form['registration[terms]']->tick();
$form['registration[terms]']->untick();

// select an option
$form['registration[birthday][year]']->select(1984);

// select many options from a "multiple" select
$form['registration[interests]']->select(array('symfony', 'cookies'));

// even fake a file upload
$form['registration[photo]']->upload('/path/to/lucas.jpg');
```

What's the point of doing all of this? If you're testing internally, you can grab the information off of your form as if it had just been submitted by using the PHP values:

Listing 35-26
```
$values = $form->getPhpValues();
$files = $form->getPhpFiles();
```

If you're using an external HTTP client, you can use the form to grab all of the information you need to create a POST request for the form:

Listing 35-27
```
$uri = $form->getUri();
$method = $form->getMethod();
$values = $form->getValues();
$files = $form->getFiles();

// now use some HTTP client and post using this information
```

One great example of an integrated system that uses all of this is *Goutte*[25]. Goutte understands the Symfony Crawler object and can use it to submit forms directly:

25. https://github.com/fabpot/goutte

Listing 35-28

```php
use Goutte\Client;

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

// select the form and fill in some values
$form = $crawler->selectButton('Log in')->form();
$form['login'] = 'symfonyfan';
$form['password'] = 'anypass';

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

Chapter 36
The EventDispatcher Component

> The EventDispatcher component provides tools that allow your application components to communicate with each other by dispatching events and listening to them.

Introduction

Object Oriented code has gone a long way to ensuring code extensibility. By creating classes that have well defined responsibilities, your code becomes more flexible and a developer can extend them with subclasses to modify their behaviors. But if they want to share the changes with other developers who have also made their own subclasses, code inheritance is no longer the answer.

Consider the real-world example where you want to provide a plugin system for your project. A plugin should be able to add methods, or do something before or after a method is executed, without interfering with other plugins. This is not an easy problem to solve with single inheritance, and multiple inheritance (were it possible with PHP) has its own drawbacks.

The Symfony EventDispatcher component implements the *Mediator*[1] pattern in a simple and effective way to make all these things possible and to make your projects truly extensible.

Take a simple example from *The HttpKernel Component*. Once a `Response` object has been created, it may be useful to allow other elements in the system to modify it (e.g. add some cache headers) before it's actually used. To make this possible, the Symfony kernel throws an event - `kernel.response`. Here's how it works:

- A *listener* (PHP object) tells a central *dispatcher* object that it wants to listen to the `kernel.response` event;
- At some point, the Symfony kernel tells the *dispatcher* object to dispatch the `kernel.response` event, passing with it an `Event` object that has access to the `Response` object;
- The dispatcher notifies (i.e. calls a method on) all listeners of the `kernel.response` event, allowing each of them to make modifications to the `Response` object.

1. `http://en.wikipedia.org/wiki/Mediator_pattern`

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/event-dispatcher` on *Packagist*[2]);
- Use the official Git repository (*https://github.com/symfony/EventDispatcher*[3]).

Usage

Events

When an event is dispatched, it's identified by a unique name (e.g. `kernel.response`), which any number of listeners might be listening to. An *`Event`*[4] instance is also created and passed to all of the listeners. As you'll see later, the `Event` object itself often contains data about the event being dispatched.

Naming Conventions

The unique event name can be any string, but optionally follows a few simple naming conventions:

- use only lowercase letters, numbers, dots (`.`), and underscores (_);
- prefix names with a namespace followed by a dot (e.g. `kernel.`);
- end names with a verb that indicates what action is being taken (e.g. `request`).

Here are some examples of good event names:

- `kernel.response`
- `form.pre_set_data`

Event Names and Event Objects

When the dispatcher notifies listeners, it passes an actual `Event` object to those listeners. The base `Event` class is very simple: it contains a method for stopping *event propagation*, but not much else.

Often times, data about a specific event needs to be passed along with the `Event` object so that the listeners have needed information. In the case of the `kernel.response` event, the `Event` object that's created and passed to each listener is actually of type *`FilterResponseEvent`*[5], a subclass of the base `Event` object. This class contains methods such as `getResponse` and `setResponse`, allowing listeners to get or even replace the `Response` object.

The moral of the story is this: When creating a listener to an event, the `Event` object that's passed to the listener may be a special subclass that has additional methods for retrieving information from and responding to the event.

The Dispatcher

The dispatcher is the central object of the event dispatcher system. In general, a single dispatcher is created, which maintains a registry of listeners. When an event is dispatched via the dispatcher, it notifies all listeners registered with that event:

Listing 36-1

2. https://packagist.org/packages/symfony/event-dispatcher
3. https://github.com/symfony/EventDispatcher
4. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html

```php
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
```

Connecting Listeners

To take advantage of an existing event, you need to connect a listener to the dispatcher so that it can be notified when the event is dispatched. A call to the dispatcher's `addListener()` method associates any valid PHP callable to an event:

Listing 36-2
```php
$listener = new AcmeListener();
$dispatcher->addListener('foo.action', array($listener, 'onFooAction'));
```

The `addListener()` method takes up to three arguments:

- The event name (string) that this listener wants to listen to;
- A PHP callable that will be notified when an event is thrown that it listens to;
- An optional priority integer (higher equals more important, and therefore that the listener will be triggered earlier) that determines when a listener is triggered versus other listeners (defaults to `0`). If two listeners have the same priority, they are executed in the order that they were added to the dispatcher.

A *PHP callable*[6] is a PHP variable that can be used by the `call_user_func()` function and returns `true` when passed to the `is_callable()` function. It can be a `\Closure` instance, an object implementing an `__invoke` method (which is what closures are in fact), a string representing a function, or an array representing an object method or a class method.

So far, you've seen how PHP objects can be registered as listeners. You can also register PHP *Closures*[7] as event listeners:

Listing 36-3
```php
use Symfony\Component\EventDispatcher\Event;

$dispatcher->addListener('foo.action', function (Event $event) {
    // will be executed when the foo.action event is dispatched
});
```

Once a listener is registered with the dispatcher, it waits until the event is notified. In the above example, when the `foo.action` event is dispatched, the dispatcher calls the `AcmeListener::onFooAction` method and passes the `Event` object as the single argument:

Listing 36-4
```php
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

    public function onFooAction(Event $event)
    {
        // ... do something
```

6. http://www.php.net/manual/en/language.pseudo-types.php#language.types.callback
7. http://php.net/manual/en/functions.anonymous.php

```
    }
}
```

In many cases, a special `Event` subclass that's specific to the given event is passed to the listener. This gives the listener access to special information about the event. Check the documentation or implementation of each event to determine the exact `Symfony\Component\EventDispatcher\Event` instance that's being passed. For example, the `kernel.response` event passes an instance of `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

Listing 36-5
```php
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    $request = $event->getRequest();

    // ...
}
```

Registering Event Listeners in the Service Container

When you are using the *`ContainerAwareEventDispatcher`*[8] and the *DependencyInjection component*, you can use the *`RegisterListenersPass`*[9] from the HttpKernel component to tag services as event listeners:

Listing 36-6
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\ParameterBag\ParameterBag;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\HttpKernel\DependencyInjection\RegisterListenersPass;

$containerBuilder = new ContainerBuilder(new ParameterBag());
$containerBuilder->addCompilerPass(new RegisterListenersPass());

// register the event dispatcher service
$containerBuilder->setDefinition('event_dispatcher', new Definition(
    'Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher',
    array(new Reference('service_container'))
));

// register your event listener service
$listener = new Definition('AcmeListener');
$listener->addTag('kernel.event_listener', array(
    'event' => 'foo.action',
    'method' => 'onFooAction',
));
$containerBuilder->setDefinition('listener_service_id', $listener);

// register an event subscriber
$subscriber = new Definition('AcmeSubscriber');
$subscriber->addTag('kernel.event_subscriber');
$containerBuilder->setDefinition('subscriber_service_id', $subscriber);
```

By default, the listeners pass assumes that the event dispatcher's service id is `event_dispatcher`, that event listeners are tagged with the `kernel.event_listener` tag and that event subscribers are tagged with the `kernel.event_subscriber` tag. You can change these default values by passing custom values to the constructor of `RegisterListenersPass`.

Creating and Dispatching an Event

In addition to registering listeners with existing events, you can create and dispatch your own events. This is useful when creating third-party libraries and also when you want to keep different components of your own system flexible and decoupled.

The Static `Events` Class

Suppose you want to create a new Event - `store.order` - that is dispatched each time an order is created inside your application. To keep things organized, start by creating a `StoreEvents` class inside your application that serves to define and document your event:

Listing 36-7
```php
namespace Acme\StoreBundle;

final class StoreEvents
```

8. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
9. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/DependencyInjection/RegisterListenersPass.html

```
4  {
5      /**
6       * The store.order event is thrown each time an order is created
7       * in the system.
8       *
9       * The event listener receives an
10      * Acme\StoreBundle\Event\FilterOrderEvent instance.
11      *
12      * @var string
13      */
14     const STORE_ORDER = 'store.order';
15 }
```

Notice that this class doesn't actually *do* anything. The purpose of the `StoreEvents` class is just to be a location where information about common events can be centralized. Notice also that a special `FilterOrderEvent` class will be passed to each listener of this event.

Creating an Event Object

Later, when you dispatch this new event, you'll create an `Event` instance and pass it to the dispatcher. The dispatcher then passes this same instance to each of the listeners of the event. If you don't need to pass any information to your listeners, you can use the default `Symfony\Component\EventDispatcher\Event` class. Most of the time, however, you *will* need to pass information about the event to each listener. To accomplish this, you'll create a new class that extends `Symfony\Component\EventDispatcher\Event`.

In this example, each listener will need access to some pretend `Order` object. Create an `Event` class that makes this possible:

Listing 36-8

```
1  namespace Acme\StoreBundle\Event;
2
3  use Symfony\Component\EventDispatcher\Event;
4  use Acme\StoreBundle\Order;
5
6  class FilterOrderEvent extends Event
7  {
8      protected $order;
9
10     public function __construct(Order $order)
11     {
12         $this->order = $order;
13     }
14
15     public function getOrder()
16     {
17         return $this->order;
18     }
19 }
```

Each listener now has access to the `Order` object via the `getOrder` method.

Dispatch the Event

The *`dispatch()`*[10] method notifies all listeners of the given event. It takes two arguments: the name of the event to dispatch and the `Event` instance to pass to each listener of that event:

Listing 36-9

10. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventDispatcher.html#dispatch()

```php
use Acme\StoreBundle\StoreEvents;
use Acme\StoreBundle\Order;
use Acme\StoreBundle\Event\FilterOrderEvent;

// the order is somehow created or retrieved
$order = new Order();
// ...

// create the FilterOrderEvent and dispatch it
$event = new FilterOrderEvent($order);
$dispatcher->dispatch(StoreEvents::STORE_ORDER, $event);
```

Notice that the special `FilterOrderEvent` object is created and passed to the `dispatch` method. Now, any listener to the `store.order` event will receive the `FilterOrderEvent` and have access to the `Order` object via the `getOrder` method:

Listing 36-10
```php
// some listener class that's been registered for "store.order" event
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    $order = $event->getOrder();
    // do something to or with the order
}
```

Using Event Subscribers

The most common way to listen to an event is to register an *event listener* with the dispatcher. This listener can listen to one or more events and is notified each time those events are dispatched.

Another way to listen to events is via an *event subscriber*. An event subscriber is a PHP class that's able to tell the dispatcher exactly which events it should subscribe to. It implements the *`EventSubscriberInterface`*[11] interface, which requires a single static method called `getSubscribedEvents`. Take the following example of a subscriber that subscribes to the `kernel.response` and `store.order` events:

Listing 36-11
```php
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class StoreSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            'kernel.response' => array(
                array('onKernelResponsePre', 10),
                array('onKernelResponseMid', 5),
                array('onKernelResponsePost', 0),
            ),
            'store.order'     => array('onStoreOrder', 0),
        );
    }
```

11. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

```
19
20      public function onKernelResponsePre(FilterResponseEvent $event)
21      {
22          // ...
23      }
24
25      public function onKernelResponseMid(FilterResponseEvent $event)
26      {
27          // ...
28      }
29
30      public function onKernelResponsePost(FilterResponseEvent $event)
31      {
32          // ...
33      }
34
35      public function onStoreOrder(FilterOrderEvent $event)
36      {
37          // ...
38      }
39  }
```

This is very similar to a listener class, except that the class itself can tell the dispatcher which events it should listen to. To register a subscriber with the dispatcher, use the *addSubscriber()*[12] method:

Listing 36-12
```
1  use Acme\StoreBundle\Event\StoreSubscriber;
2
3  $subscriber = new StoreSubscriber();
4  $dispatcher->addSubscriber($subscriber);
```

The dispatcher will automatically register the subscriber for each event returned by the `getSubscribedEvents` method. This method returns an array indexed by event names and whose values are either the method name to call or an array composed of the method name to call and a priority. The example above shows how to register several listener methods for the same event in subscriber and also shows how to pass the priority of each listener method. The higher the priority, the earlier the method is called. In the above example, when the `kernel.response` event is triggered, the methods `onKernelResponsePre`, `onKernelResponseMid`, and `onKernelResponsePost` are called in that order.

Stopping Event Flow/Propagation

In some cases, it may make sense for a listener to prevent any other listeners from being called. In other words, the listener needs to be able to tell the dispatcher to stop all propagation of the event to future listeners (i.e. to not notify any more listeners). This can be accomplished from inside a listener via the *stopPropagation()*[13] method:

Listing 36-13
```
1  use Acme\StoreBundle\Event\FilterOrderEvent;
2
3  public function onStoreOrder(FilterOrderEvent $event)
4  {
5      // ...
6
7      $event->stopPropagation();
8  }
```

12. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventDispatcher.html#addSubscriber()
13. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html#stopPropagation()

Now, any listeners to `store.order` that have not yet been called will *not* be called.

It is possible to detect if an event was stopped by using the *`isPropagationStopped()`*[14] method which returns a boolean value:

Listing 36-14
```php
$dispatcher->dispatch('foo.event', $event);
if ($event->isPropagationStopped()) {
    // ...
}
```

EventDispatcher aware Events and Listeners

The `EventDispatcher` always injects a reference to itself in the passed event object. This means that all listeners have direct access to the `EventDispatcher` object that notified the listener via the passed `Event` object's *`getDispatcher()`*[15] method.

This can lead to some advanced applications of the `EventDispatcher` including letting listeners dispatch other events, event chaining or even lazy loading of more listeners into the dispatcher object. Examples follow:

Lazy loading listeners:

Listing 36-15
```php
use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Event\StoreSubscriber;

class Foo
{
    private $started = false;

    public function myLazyListener(Event $event)
    {
        if (false === $this->started) {
            $subscriber = new StoreSubscriber();
            $event->getDispatcher()->addSubscriber($subscriber);
        }

        $this->started = true;

        // ... more code
    }
}
```

Dispatching another event from within a listener:

Listing 36-16
```php
use Symfony\Component\EventDispatcher\Event;

class Foo
{
    public function myFooListener(Event $event)
    {
        $event->getDispatcher()->dispatch('log', $event);

        // ... more code
    }
}
```

14. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html#isPropagationStopped()
15. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html#getDispatcher()

While this above is sufficient for most uses, if your application uses multiple `EventDispatcher` instances, you might need to specifically inject a known instance of the `EventDispatcher` into your listeners. This could be done using constructor or setter injection as follows:

Constructor injection:

Listing 36-17
```php
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class Foo
{
    protected $dispatcher = null;

    public function __construct(EventDispatcherInterface $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Or setter injection:

Listing 36-18
```php
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class Foo
{
    protected $dispatcher = null;

    public function setEventDispatcher(EventDispatcherInterface $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Choosing between the two is really a matter of taste. Many tend to prefer the constructor injection as the objects are fully initialized at construction time. But when you have a long list of dependencies, using setter injection can be the way to go, especially for optional dependencies.

Dispatcher Shortcuts

The *`EventDispatcher::dispatch`*[16] method always returns an *`Event`*[17] object. This allows for various shortcuts. For example, if one does not need a custom event object, one can simply rely on a plain *`Event`*[18] object. You do not even need to pass this to the dispatcher as it will create one by default unless you specifically pass one:

Listing 36-19
```php
$dispatcher->dispatch('foo.event');
```

Moreover, the EventDispatcher always returns whichever event object that was dispatched, i.e. either the event that was passed or the event that was created internally by the dispatcher. This allows for nice shortcuts:

Listing 36-20
```php
if (!$dispatcher->dispatch('foo.event')->isPropagationStopped()) {
    // ...
}
```

16. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventDispatcher.html#dispatch()
17. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html
18. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html

Or:

Listing 36-21
```php
$barEvent = new BarEvent();
$bar = $dispatcher->dispatch('bar.event', $barEvent)->getBar();
```

Or:

Listing 36-22
```php
$bar = $dispatcher->dispatch('bar.event', new BarEvent())->getBar();
```

and so on...

Event Name Introspection

Since the `EventDispatcher` already knows the name of the event when dispatching it, the event name is also injected into the *`Event`*[19] objects, making it available to event listeners via the *`getName()`*[20] method.

The event name, (as with any other data in a custom event object) can be used as part of the listener's processing logic:

Listing 36-23
```php
use Symfony\Component\EventDispatcher\Event;

class Foo
{
    public function myEventListener(Event $event)
    {
        echo $event->getName();
    }
}
```

Other Dispatchers

Besides the commonly used `EventDispatcher`, the component comes with 2 other dispatchers:

- *The Container Aware Event Dispatcher*
- *The Immutable Event Dispatcher*

19. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html
20. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html#getName()

Chapter 37

The Container Aware Event Dispatcher

Introduction

The *ContainerAwareEventDispatcher*[1] is a special EventDispatcher implementation which is coupled to the service container that is part of *the DependencyInjection component*. It allows services to be specified as event listeners making the EventDispatcher extremely powerful.

Services are lazy loaded meaning the services attached as listeners will only be created if an event is dispatched that requires those listeners.

Setup

Setup is straightforward by injecting a *ContainerInterface*[2] into the *ContainerAwareEventDispatcher*[3]:

Listing 37-1
```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher;

$container = new ContainerBuilder();
$dispatcher = new ContainerAwareEventDispatcher($container);
```

Adding Listeners

The *Container Aware EventDispatcher* can either load specified services directly, or services that implement *EventSubscriberInterface*[4].

1. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
2. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/ContainerInterface.html
3. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
4. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

The following examples assume the service container has been loaded with any services that are mentioned.

Services must be marked as public in the container.

Adding Services

To connect existing service definitions, use the *`addListenerService()`*[5] method where the `$callback` is an array of `array($serviceId, $methodName)`:

Listing 37-2
```
1 $dispatcher->addListenerService($eventName, array('foo', 'logListener'));
```

Adding Subscriber Services

`EventSubscribers` can be added using the *`addSubscriberService()`*[6] method where the first argument is the service ID of the subscriber service, and the second argument is the service's class name (which must implement *`EventSubscriberInterface`*[7]) as follows:

Listing 37-3
```
1 $dispatcher->addSubscriberService(
2     'kernel.store_subscriber',
3     'StoreSubscriber'
4 );
```

The `EventSubscriberInterface` will be exactly as you would expect:

Listing 37-4
```
1  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
2  // ...
3
4  class StoreSubscriber implements EventSubscriberInterface
5  {
6      public static function getSubscribedEvents()
7      {
8          return array(
9              'kernel.response' => array(
10                 array('onKernelResponsePre', 10),
11                 array('onKernelResponsePost', 0),
12             ),
13             'store.order'     => array('onStoreOrder', 0),
14         );
15     }
16
17     public function onKernelResponsePre(FilterResponseEvent $event)
18     {
19         // ...
20     }
21
22     public function onKernelResponsePost(FilterResponseEvent $event)
23     {
24         // ...
```

5. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#addListenerService()
6. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#addSubscriberService()
7. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

```
25      }
26
27      public function onStoreOrder(FilterOrderEvent $event)
28      {
29          // ...
30      }
31  }
```

Chapter 38
The Generic Event Object

The base *Event*[1] class provided by the EventDispatcher component is deliberately sparse to allow the creation of API specific event objects by inheritance using OOP. This allows for elegant and readable code in complex applications.

The *GenericEvent*[2] is available for convenience for those who wish to use just one event object throughout their application. It is suitable for most purposes straight out of the box, because it follows the standard observer pattern where the event object encapsulates an event 'subject', but has the addition of optional extra arguments.

GenericEvent[3] has a simple API in addition to the base class *Event*[4]

- *__construct()*[5]: Constructor takes the event subject and any arguments;
- *getSubject()*[6]: Get the subject;
- *setArgument()*[7]: Sets an argument by key;
- *setArguments()*[8]: Sets arguments array;
- *getArgument()*[9]: Gets an argument by key;
- *getArguments()*[10]: Getter for all arguments;
- *hasArgument()*[11]: Returns true if the argument key exists;

The `GenericEvent` also implements *ArrayAccess*[12] on the event arguments which makes it very convenient to pass extra arguments regarding the event subject.

The following examples show use-cases to give a general idea of the flexibility. The examples assume event listeners have been added to the dispatcher.

Simply passing a subject:

---

1. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html
2. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html
3. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html
4. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#__construct()
6. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#getSubject()
7. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#setArgument()
8. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#setArguments()
9. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#getArgument()
10. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#getArguments()
11. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/GenericEvent.html#hasArgument()
12. http://php.net/manual/en/class.arrayaccess.php

Listing 38-1
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject);
$dispatcher->dispatch('foo', $event);

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if ($event->getSubject() instanceof Foo) {
            // ...
        }
    }
}
```

Passing and processing arguments using the *ArrayAccess*[13] API to access the event arguments:

Listing 38-2
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent(
    $subject,
    array('type' => 'foo', 'counter' => 0)
);
$dispatcher->dispatch('foo', $event);

echo $event['counter'];

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if (isset($event['type']) && $event['type'] === 'foo') {
            // ... do something
        }

        $event['counter']++;
    }
}
```

Filtering data:

Listing 38-3
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject, array('data' => 'Foo'));
$dispatcher->dispatch('foo', $event);

echo $event['data'];

class FooListener
{
    public function filter(GenericEvent $event)
    {
        $event['data'] = strtolower($event['data']);
    }
}
```

13. http://php.net/manual/en/class.arrayaccess.php

Chapter 39
The Immutable Event Dispatcher

New in version 2.1: This feature was introduced in Symfony 2.1.

The *`ImmutableEventDispatcher`*[1] is a locked or frozen event dispatcher. The dispatcher cannot register new listeners or subscribers.

The `ImmutableEventDispatcher` takes another event dispatcher with all the listeners and subscribers. The immutable dispatcher is just a proxy of this original dispatcher.

To use it, first create a normal dispatcher (`EventDispatcher` or `ContainerAwareEventDispatcher`) and register some listeners or subscribers:

Listing 39-1
```php
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
$dispatcher->addListener('foo.action', function ($event) {
    // ...
});

// ...
```

Now, inject that into an `ImmutableEventDispatcher`:

Listing 39-2
```php
use Symfony\Component\EventDispatcher\ImmutableEventDispatcher;
// ...

$immutableDispatcher = new ImmutableEventDispatcher($dispatcher);
```

You'll need to use this new dispatcher in your project.

If you are trying to execute one of the methods which modifies the dispatcher (e.g. `addListener`), a `BadMethodCallException` is thrown.

1. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/ImmutableEventDispatcher.html

Chapter 40
The Traceable Event Dispatcher

The *`TraceableEventDispatcher`*[1] is an event dispatcher that wraps any other event dispatcher and can then be used to determine which event listeners have been called by the dispatcher. Pass the event dispatcher to be wrapped and an instance of the *`Stopwatch`*[2] to its constructor:

Listing 40-1
```php
use Symfony\Component\HttpKernel\Debug\TraceableEventDispatcher;
use Symfony\Component\Stopwatch\Stopwatch;

// the event dispatcher to debug
$eventDispatcher = ...;

$traceableEventDispatcher = new TraceableEventDispatcher(
    $eventDispatcher,
    new Stopwatch()
);
```

Now, the `TraceableEventDispatcher` can be used like any other event dispatcher to register event listeners and dispatch events:

Listing 40-2
```php
// ...

// register an event listener
$eventListener = ...;
$priority = ...;
$traceableEventDispatcher->addListener(
    'event.the_name',
    $eventListener,
    $priority
);

// dispatch an event
$event = ...;
$traceableEventDispatcher->dispatch('event.the_name', $event);
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Debug/TraceableEventDispatcher.html
2. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html

After your application has been processed, you can use the *getCalledListeners()*[3] method to retrieve an array of event listeners that have been called in your application. Similarly, the *getNotCalledListeners()*[4] method returns an array of event listeners that have not been called:

Listing 40-3
```
// ...

$calledListeners = $traceableEventDispatcher->getCalledListeners();
$notCalledListeners = $traceableEventDispatcher->getNotCalledListeners();
```

3. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Debug/TraceableEventDispatcherInterface.html#getCalledListeners()
4. http://api.symfony.com/2.3/Symfony/Component/EventDispatcher/Debug/TraceableEventDispatcherInterface.html#getNotCalledListeners()

Chapter 41
The Filesystem Component

> The Filesystem component provides basic utilities for the filesystem.

New in version 2.1: The Filesystem component was introduced in Symfony 2.1. Previously, the `Filesystem` class was located in the HttpKernel component.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/filesystem` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Filesystem*[2]).

Usage

The *`Filesystem`*[3] class is the unique endpoint for filesystem operations:

Listing 41-1
```php
use Symfony\Component\Filesystem\Filesystem;
use Symfony\Component\Filesystem\Exception\IOException;

$fs = new Filesystem();

try {
    $fs->mkdir('/tmp/random/dir/'.mt_rand());
} catch (IOException $e) {
    echo "An error occurred while creating your directory";
}
```

1. https://packagist.org/packages/symfony/filesystem
2. https://github.com/symfony/Filesystem
3. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html

Methods *mkdir()*[4], *exists()*[5], *touch()*[6], *remove()*[7], *chmod()*[8], *chown()*[9] and *chgrp()*[10] can receive a string, an array or any object implementing *Traversable*[11] as the target argument.

mkdir

mkdir()[12] creates a directory. On POSIX filesystems, directories are created with a default mode value *0777*. You can use the second argument to set your own mode:

Listing 41-2
```
1 $fs->mkdir('/tmp/photos', 0700);
```

You can pass an array or any *Traversable*[13] object as the first argument.

exists

exists()[14] checks for the presence of all files or directories and returns `false` if a file is missing:

Listing 41-3
```
1 // this directory exists, return true
2 $fs->exists('/tmp/photos');
3
4 // rabbit.jpg exists, bottle.png does not exists, return false
5 $fs->exists(array('rabbit.jpg', 'bottle.png'));
```

You can pass an array or any *Traversable*[15] object as the first argument.

copy

copy()[16] is used to copy files. If the target already exists, the file is copied only if the source modification date is later than the target. This behavior can be overridden by the third boolean argument:

Listing 41-4
```
1 // works only if image-ICC has been modified after image.jpg
2 $fs->copy('image-ICC.jpg', 'image.jpg');
3
```

4. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#mkdir()
5. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#exists()
6. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#touch()
7. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#remove()
8. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chmod()
9. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chown()
10. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chgrp()
11. http://php.net/manual/en/class.traversable.php
12. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#mkdir()
13. http://php.net/manual/en/class.traversable.php
14. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#exists()
15. http://php.net/manual/en/class.traversable.php
16. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#copy()

```
4 // image.jpg will be overridden
5 $fs->copy('image-ICC.jpg', 'image.jpg', true);
```

touch

touch()[17] sets access and modification time for a file. The current time is used by default. You can set your own with the second argument. The third argument is the access time:

Listing 41-5
```
1 // set modification time to the current timestamp
2 $fs->touch('file.txt');
3 // set modification time 10 seconds in the future
4 $fs->touch('file.txt', time() + 10);
5 // set access time 10 seconds in the past
6 $fs->touch('file.txt', time(), time() - 10);
```

You can pass an array or any *Traversable*[18] object as the first argument.

chown

chown()[19] is used to change the owner of a file. The third argument is a boolean recursive option:

Listing 41-6
```
1 // set the owner of the lolcat video to www-data
2 $fs->chown('lolcat.mp4', 'www-data');
3 // change the owner of the video directory recursively
4 $fs->chown('/video', 'www-data', true);
```

You can pass an array or any *Traversable*[20] object as the first argument.

chgrp

chgrp()[21] is used to change the group of a file. The third argument is a boolean recursive option:

Listing 41-7
```
1 // set the group of the lolcat video to nginx
2 $fs->chgrp('lolcat.mp4', 'nginx');
3 // change the group of the video directory recursively
4 $fs->chgrp('/video', 'nginx', true);
```

17. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#touch()
18. http://php.net/manual/en/class.traversable.php
19. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chown()
20. http://php.net/manual/en/class.traversable.php
21. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chgrp()

You can pass an array or any *Traversable*[22] object as the first argument.

chmod

chmod()[23] is used to change the mode of a file. The fourth argument is a boolean recursive option:

Listing 41-8
```
1 // set the mode of the video to 0600
2 $fs->chmod('video.ogg', 0600);
3 // change the mod of the src directory recursively
4 $fs->chmod('src', 0700, 0000, true);
```

You can pass an array or any *Traversable*[24] object as the first argument.

remove

remove()[25] is used to remove files, symlinks, directories easily:

Listing 41-9
```
1 $fs->remove(array('symlink', '/path/to/directory', 'activity.log'));
```

You can pass an array or any *Traversable*[26] object as the first argument.

rename

rename()[27] is used to rename files and directories:

Listing 41-10
```
1 // rename a file
2 $fs->rename('/tmp/processed_video.ogg', '/path/to/store/video_647.ogg');
3 // rename a directory
4 $fs->rename('/tmp/files', '/path/to/store/files');
```

symlink

symlink()[28] creates a symbolic link from the target to the destination. If the filesystem does not support symbolic links, a third boolean argument is available:

Listing 41-11

22. http://php.net/manual/en/class.traversable.php
23. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#chmod()
24. http://php.net/manual/en/class.traversable.php
25. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#remove()
26. http://php.net/manual/en/class.traversable.php
27. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#rename()
28. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#symlink()

```php
// create a symbolic link
$fs->symlink('/path/to/source', '/path/to/destination');
// duplicate the source directory if the filesystem
// does not support symbolic links
$fs->symlink('/path/to/source', '/path/to/destination', true);
```

makePathRelative

makePathRelative()[29] returns the relative path of a directory given another one:

Listing 41-12
```php
// returns '../'
$fs->makePathRelative(
    '/var/lib/symfony/src/Symfony/',
    '/var/lib/symfony/src/Symfony/Component'
);
// returns 'videos/'
$fs->makePathRelative('/tmp/videos', '/tmp')
```

mirror

mirror()[30] mirrors a directory:

Listing 41-13
```php
$fs->mirror('/path/to/source', '/path/to/target');
```

isAbsolutePath

isAbsolutePath()[31] returns `true` if the given path is absolute, `false` otherwise:

Listing 41-14
```php
// return true
$fs->isAbsolutePath('/tmp');
// return true
$fs->isAbsolutePath('c:\\Windows');
// return false
$fs->isAbsolutePath('tmp');
// return false
$fs->isAbsolutePath('../dir');
```

dumpFile

New in version 2.3: The `dumpFile()` was introduced in Symfony 2.3.

dumpFile()[32] allows you to dump contents to a file. It does this in an atomic manner: it writes a temporary file first and then moves it to the new file location when it's finished. This means that the user will always see either the complete old file or complete new file (but never a partially-written file):

Listing 41-15
```php
$fs->dumpFile('file.txt', 'Hello World');
```

29. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#makePathRelative()
30. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#mirror()
31. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#isAbsolutePath()
32. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Filesystem.html#dumpFile()

The `file.txt` file contains `Hello World` now.

A desired file mode can be passed as the third argument.

Error Handling

Whenever something wrong happens, an exception implementing *`ExceptionInterface`*[33] is thrown.

Prior to version 2.1, `mkdir` returned a boolean and did not throw exceptions. As of 2.1, a *`IOException`*[34] is thrown if a directory creation fails.

33. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Exception/ExceptionInterface.html
34. http://api.symfony.com/2.3/Symfony/Component/Filesystem/Exception/IOException.html

Chapter 42
The Finder Component

> The Finder component finds files and directories via an intuitive fluent interface.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/finder` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Finder*[2]).

Usage

The *`Finder`*[3] class finds files and/or directories:

Listing 42-1
```php
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    // Print the absolute path
    print $file->getRealpath()."\n";

    // Print the relative path to the file, omitting the filename
    print $file->getRelativePath()."\n";

    // Print the relative path to the file
```

1. https://packagist.org/packages/symfony/finder
2. https://github.com/symfony/Finder
3. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html

```
14      print $file->getRelativePathname()."\n";
15  }
```

The `$file` is an instance of *SplFileInfo*[4] which extends *SplFileInfo*[5] to provide methods to work with relative paths.

The above code prints the names of all the files in the current directory recursively. The Finder class uses a fluent interface, so all methods return the Finder instance.

A Finder instance is a PHP *Iterator*[6]. So, instead of iterating over the Finder with `foreach`, you can also convert it to an array with the *iterator_to_array*[7] method, or get the number of items with *iterator_count*[8].

When searching through multiple locations passed to the *in()*[9] method, a separate iterator is created internally for every location. This means we have multiple result sets aggregated into one. Since *iterator_to_array*[10] uses keys of result sets by default, when converting to an array, some keys might be duplicated and their values overwritten. This can be avoided by passing `false` as a second parameter to *iterator_to_array*[11].

Criteria

There are lots of ways to filter and sort your results.

Location

The location is the only mandatory criteria. It tells the finder which directory to use for the search:

Listing 42-2
```
1  $finder->in(__DIR__);
```

Search in several locations by chaining calls to *in()*[12]:

Listing 42-3
```
1  $finder->files()->in(__DIR__)->in('/elsewhere');
```

New in version 2.2: Wildcard support was introduced in version 2.2.

Use wildcard characters to search in the directories matching a pattern:

Listing 42-4
```
1  $finder->in('src/Symfony/*/*/Resources');
```

Each pattern has to resolve to at least one directory path.

Exclude directories from matching with the *exclude()*[13] method:

4. http://api.symfony.com/2.3/Symfony/Component/Finder/SplFileInfo.html
5. http://php.net/manual/en/class.splfileinfo.php
6. http://php.net/manual/en/class.iterator.php
7. http://php.net/manual/en/function.iterator-to-array.php
8. http://php.net/manual/en/function.iterator-count.php
9. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#in()
10. http://php.net/manual/en/function.iterator-to-array.php
11. http://php.net/manual/en/function.iterator-to-array.php
12. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#in()
13. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#exclude()

Listing 42-5
```php
$finder->in(__DIR__)->exclude('ruby');
```

New in version 2.3: The *`ignoreUnreadableDirs()`*[14] method was introduced in Symfony 2.3.
It's also possible to ignore directories that you don't have permission to read:

Listing 42-6
```php
$finder->ignoreUnreadableDirs()->in(__DIR__);
```

As the Finder uses PHP iterators, you can pass any URL with a supported *protocol*[15]:

Listing 42-7
```php
$finder->in('ftp://example.com/pub/');
```

And it also works with user-defined streams:

Listing 42-8
```php
use Symfony\Component\Finder\Finder;

$s3 = new \Zend_Service_Amazon_S3($key, $secret);
$s3->registerStreamWrapper("s3");

$finder = new Finder();
$finder->name('photos*')->size('< 100K')->date('since 1 hour ago');
foreach ($finder->in('s3://bucket-name') as $file) {
    // ... do something

    print $file->getFilename()."\n";
}
```

Read the *Streams*[16] documentation to learn how to create your own streams.

Files or Directories

By default, the Finder returns files and directories; but the *`files()`*[17] and *`directories()`*[18] methods control that:

Listing 42-9
```php
$finder->files();

$finder->directories();
```

If you want to follow links, use the `followLinks()` method:

Listing 42-10
```php
$finder->files()->followLinks();
```

By default, the iterator ignores popular VCS files. This can be changed with the `ignoreVCS()` method:

Listing 42-11

14. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#ignoreUnreadableDirs()
15. http://www.php.net/manual/en/wrappers.php
16. http://www.php.net/streams
17. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#files()
18. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#directories()

```php
$finder->ignoreVCS(false);
```

Sorting

Sort the result by name or by type (directories first, then files):

Listing 42-12
```php
$finder->sortByName();

$finder->sortByType();
```

Notice that the `sort*` methods need to get all matching elements to do their jobs. For large iterators, it is slow.

You can also define your own sorting algorithm with `sort()` method:

Listing 42-13
```php
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

File Name

Restrict files by name with the *name()*[19] method:

Listing 42-14
```php
$finder->files()->name('*.php');
```

The `name()` method accepts globs, strings, or regexes:

Listing 42-15
```php
$finder->files()->name('/\.php$/');
```

The `notName()` method excludes files matching a pattern:

Listing 42-16
```php
$finder->files()->notName('*.rb');
```

File Contents

Restrict files by contents with the *contains()*[20] method:

Listing 42-17
```php
$finder->files()->contains('lorem ipsum');
```

The `contains()` method accepts strings or regexes:

Listing 42-18

19. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#name()
20. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#contains()

```php
$finder->files()->contains('/lorem\s+ipsum$/i');
```

The `notContains()` method excludes files containing given pattern:

Listing 42-19
```php
$finder->files()->notContains('dolor sit amet');
```

Path

New in version 2.2: The `path()` and `notPath()` methods were introduced in Symfony 2.2.

Restrict files and directories by path with the *`path()`*[21] method:

Listing 42-20
```php
$finder->path('some/special/dir');
```

On all platforms slash (i.e. `/`) should be used as the directory separator.

The `path()` method accepts a string or a regular expression:

Listing 42-21
```php
$finder->path('foo/bar');
$finder->path('/^foo\/bar/');
```

Internally, strings are converted into regular expressions by escaping slashes and adding delimiters:

Listing 42-22
```
dirname    ===>    /dirname/
a/b/c      ===>    /a\/b\/c/
```

The *`notPath()`*[22] method excludes files by path:

Listing 42-23
```php
$finder->notPath('other/dir');
```

File Size

Restrict files by size with the *`size()`*[23] method:

Listing 42-24
```php
$finder->files()->size('< 1.5K');
```

Restrict by a size range by chaining calls:

Listing 42-25
```php
$finder->files()->size('>= 1K')->size('<= 2K');
```

The comparison operator can be any of the following: `>`, `>=`, `<`, `<=`, `==`, `!=`.

The target value may use magnitudes of kilobytes (`k`, `ki`), megabytes (`m`, `mi`), or gigabytes (`g`, `gi`). Those suffixed with an `i` use the appropriate `2**n` version in accordance with the *IEC standard*[24].

File Date

Restrict files by last modified dates with the *`date()`*[25] method:

21. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#path()
22. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#notPath()
23. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#size()
24. http://physics.nist.gov/cuu/Units/binary.html

Listing 42-26
```
$finder->date('since yesterday');
```

The comparison operator can be any of the following: `>`, `>=`, `<`, `<=`, `==`. You can also use `since` or `after` as an alias for `>`, and `until` or `before` as an alias for `<`.

The target value can be any date supported by the *strtotime*[26] function.

Directory Depth

By default, the Finder recursively traverse directories. Restrict the depth of traversing with *depth()*[27]:

Listing 42-27
```
$finder->depth('== 0');
$finder->depth('< 3');
```

Custom Filtering

To restrict the matching file with your own strategy, use *filter()*[28]:

Listing 42-28
```
$filter = function (\SplFileInfo $file)
{
    if (strlen($file) > 10) {
        return false;
    }
};

$finder->files()->filter($filter);
```

The `filter()` method takes a Closure as an argument. For each matching file, it is called with the file as a *SplFileInfo*[29] instance. The file is excluded from the result set if the Closure returns `false`.

Reading Contents of Returned Files

The contents of returned files can be read with *getContents()*[30]:

Listing 42-29
```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    $contents = $file->getContents();

    // ...
}
```

25. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#date()
26. http://www.php.net/manual/en/datetime.formats.php
27. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#depth()
28. http://api.symfony.com/2.3/Symfony/Component/Finder/Finder.html#filter()
29. http://api.symfony.com/2.3/Symfony/Component/Finder/SplFileInfo.html
30. http://api.symfony.com/2.3/Symfony/Component/Finder/SplFileInfo.html#getContents()

Chapter 43
The Form Component

> The Form component allows you to easily create, process and reuse HTML forms.

The Form component is a tool to help you solve the problem of allowing end-users to interact with the data and modify the data in your application. And though traditionally this has been through HTML forms, the component focuses on processing data to and from your client and application, whether that data be from a normal form post or from an API.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/form` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Form*[2]).

Configuration

If you are working with the full-stack Symfony framework, the Form component is already configured for you. In this case, skip to *Creating a simple Form*.

In Symfony, forms are represented by objects and these objects are built by using a *form factory*. Building a form factory is simple:

Listing 43-1

1. `https://packagist.org/packages/symfony/form`
2. `https://github.com/symfony/Form`

```php
use Symfony\Component\Form\Forms;

$formFactory = Forms::createFormFactory();
```

This factory can already be used to create basic forms, but it is lacking support for very important features:

- **Request Handling:** Support for request handling and file uploads;
- **CSRF Protection:** Support for protection against Cross-Site-Request-Forgery (CSRF) attacks;
- **Templating:** Integration with a templating layer that allows you to reuse HTML fragments when rendering a form;
- **Translation:** Support for translating error messages, field labels and other strings;
- **Validation:** Integration with a validation library to generate error messages for submitted data.

The Symfony Form component relies on other libraries to solve these problems. Most of the time you will use Twig and the Symfony *HttpFoundation*, Translation and Validator components, but you can replace any of these with a different library of your choice.

The following sections explain how to plug these libraries into the form factory.

For a working example, see *https://github.com/bschussek/standalone-forms*[3]

Request Handling

New in version 2.3: The `handleRequest()` method was introduced in Symfony 2.3.

To process form data, you'll need to call the *`handleRequest()`*[4] method:

Listing 43-2
```php
$form->handleRequest();
```

Behind the scenes, this uses a *`NativeRequestHandler`*[5] object to read data off of the correct PHP superglobals (i.e. `$_POST` or `$_GET`) based on the HTTP method configured on the form (POST is default).

> *If you need more control over exactly when your form is submitted or which data is passed to it, you can use the `submit()`*[6] *for this. Read more about it in the cookbook.*

3. https://github.com/bschussek/standalone-forms
4. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#handleRequest()
5. http://api.symfony.com/2.3/Symfony/Component/Form/NativeRequestHandler.html
6. *http://api.symfony.com/2.3/Symfony/Component/Form/FormInterface.html#submit()*

Integration with the HttpFoundation Component

If you use the HttpFoundation component, then you should add the *HttpFoundationExtension*[7] to your form factory:

Listing 43-3
```php
use Symfony\Component\Form\Forms;
use Symfony\Component\Form\Extension\HttpFoundation\HttpFoundationExtension;

$formFactory = Forms::createFormFactoryBuilder()
    ->addExtension(new HttpFoundationExtension())
    ->getFormFactory();
```

Now, when you process a form, you can pass the *Request*[8] object to *handleRequest()*[9]:

Listing 43-4
```php
$form->handleRequest($request);
```

For more information about the HttpFoundation component or how to install it, see *The HttpFoundation Component*.

CSRF Protection

Protection against CSRF attacks is built into the Form component, but you need to explicitly enable it or replace it with a custom solution. The following snippet adds CSRF protection to the form factory:

Listing 43-5
```php
use Symfony\Component\Form\Forms;
use Symfony\Component\Form\Extension\Csrf\CsrfExtension;
use Symfony\Component\Form\Extension\Csrf\CsrfProvider\SessionCsrfProvider;
use Symfony\Component\HttpFoundation\Session\Session;

// generate a CSRF secret from somewhere
$csrfSecret = '<generated token>';

// create a Session object from the HttpFoundation component
$session = new Session();

$csrfProvider = new SessionCsrfProvider($session, $csrfSecret);

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->addExtension(new CsrfExtension($csrfProvider))
    ->getFormFactory();
```

To secure your application against CSRF attacks, you need to define a CSRF secret. Generate a random string with at least 32 characters, insert it in the above snippet and make sure that nobody except your web server can access the secret.

Internally, this extension will automatically add a hidden field to every form (called `__token` by default) whose value is automatically generated and validated when binding the form.

7. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/HttpFoundation/HttpFoundationExtension.html
8. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
9. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#handleRequest()

If you're not using the HttpFoundation component, you can use *DefaultCsrfProvider*[10] instead, which relies on PHP's native session handling:

Listing 43-6
```php
use Symfony\Component\Form\Extension\Csrf\CsrfProvider\DefaultCsrfProvider;

$csrfProvider = new DefaultCsrfProvider($csrfSecret);
```

Twig Templating

If you're using the Form component to process HTML forms, you'll need a way to easily render your form as HTML form fields (complete with field values, errors, and labels). If you use *Twig*[11] as your template engine, the Form component offers a rich integration.

To use the integration, you'll need the `TwigBridge`, which provides integration between Twig and several Symfony components. If you're using Composer, you could install the latest 2.3 version by adding the following `require` line to your `composer.json` file:

Listing 43-7
```json
{
    "require": {
        "symfony/twig-bridge": "2.3.*"
    }
}
```

The TwigBridge integration provides you with several *Twig Functions* that help you render the HTML widget, label and error for each field (as well as a few other things). To configure the integration, you'll need to bootstrap or access Twig and add the *FormExtension*[12]:

Listing 43-8
```php
use Symfony\Component\Form\Forms;
use Symfony\Bridge\Twig\Extension\FormExtension;
use Symfony\Bridge\Twig\Form\TwigRenderer;
use Symfony\Bridge\Twig\Form\TwigRendererEngine;

// the Twig file that holds all the default markup for rendering forms
// this file comes with TwigBridge
$defaultFormTheme = 'form_div_layout.html.twig';

$vendorDir = realpath(__DIR__.'/../vendor');
// the path to TwigBridge so Twig can locate the
// form_div_layout.html.twig file
$vendorTwigBridgeDir =
    $vendorDir.'/symfony/twig-bridge/Symfony/Bridge/Twig';
// the path to your other templates
$viewsDir = realpath(__DIR__.'/../views');

$twig = new Twig_Environment(new Twig_Loader_Filesystem(array(
    $viewsDir,
    $vendorTwigBridgeDir.'/Resources/views/Form',
)));
$formEngine = new TwigRendererEngine(array($defaultFormTheme));
$formEngine->setEnvironment($twig);
// add the FormExtension to Twig
```

10. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Csrf/CsrfProvider/DefaultCsrfProvider.html
11. http://twig.sensiolabs.org
12. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Extension/FormExtension.html

```php
$twig->addExtension(
    new FormExtension(new TwigRenderer($formEngine, $csrfProvider))
);

// create your form factory as normal
$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->getFormFactory();
```

The exact details of your *Twig Configuration*[13] will vary, but the goal is always to add the *FormExtension*[14] to Twig, which gives you access to the Twig functions for rendering forms. To do this, you first need to create a *TwigRendererEngine*[15], where you define your *form themes* (i.e. resources/files that define form HTML markup).

For general details on rendering forms, see *How to Customize Form Rendering*.

If you use the Twig integration, read "*Translation*" below for details on the needed translation filters.

Translation

If you're using the Twig integration with one of the default form theme files (e.g. `form_div_layout.html.twig`), there are 2 Twig filters (`trans` and `transChoice`) that are used for translating form labels, errors, option text and other strings.

To add these Twig filters, you can either use the built-in *TranslationExtension*[16] that integrates with Symfony's Translation component, or add the 2 Twig filters yourself, via your own Twig extension.

To use the built-in integration, be sure that your project has Symfony's Translation and *Config* components installed. If you're using Composer, you could get the latest 2.3 version of each of these by adding the following to your `composer.json` file:

Listing 43-9
```json
{
    "require": {
        "symfony/translation": "2.3.*",
        "symfony/config": "2.3.*"
    }
}
```

Next, add the *TranslationExtension*[17] to your `Twig_Environment` instance:

Listing 43-10
```php
use Symfony\Component\Form\Forms;
use Symfony\Component\Translation\Translator;
use Symfony\Component\Translation\Loader\XliffFileLoader;
use Symfony\Bridge\Twig\Extension\TranslationExtension;

// create the Translator
$translator = new Translator('en');
// somehow load some translations into it
```

13. http://twig.sensiolabs.org/doc/intro.html
14. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Extension/FormExtension.html
15. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Form/TwigRendererEngine.html
16. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Extension/TranslationExtension.html
17. http://api.symfony.com/2.3/Symfony/Bridge/Twig/Extension/TranslationExtension.html

```php
$translator->addLoader('xlf', new XliffFileLoader());
$translator->addResource(
    'xlf',
    __DIR__.'/path/to/translations/messages.en.xlf',
    'en'
);

// add the TranslationExtension (gives us trans and transChoice filters)
$twig->addExtension(new TranslationExtension($translator));

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->getFormFactory();
```

Depending on how your translations are being loaded, you can now add string keys, such as field labels, and their translations to your translation files.

For more details on translations, see *Translations*.

Validation

The Form component comes with tight (but optional) integration with Symfony's Validator component. If you're using a different solution for validation, no problem! Simply take the submitted/bound data of your form (which is an array or object) and pass it through your own validation system.

To use the integration with Symfony's Validator component, first make sure it's installed in your application. If you're using Composer and want to install the latest 2.3 version, add this to your `composer.json`:

Listing 43-11
```json
{
    "require": {
        "symfony/validator": "2.3.*"
    }
}
```

If you're not familiar with Symfony's Validator component, read more about it: *Validation*. The Form component comes with a *`ValidatorExtension`*[18] class, which automatically applies validation to your data on bind. These errors are then mapped to the correct field and rendered.

Your integration with the Validation component will look something like this:

Listing 43-12
```php
use Symfony\Component\Form\Forms;
use Symfony\Component\Form\Extension\Validator\ValidatorExtension;
use Symfony\Component\Validator\Validation;

$vendorDir = realpath(__DIR__.'/../vendor');
$vendorFormDir = $vendorDir.'/symfony/form/Symfony/Component/Form';
$vendorValidatorDir =
    $vendorDir.'/symfony/validator/Symfony/Component/Validator';

// create the validator - details will vary
$validator = Validation::createValidator();

// there are built-in translations for the core error messages
$translator->addResource(
    'xlf',
```

18. http://api.symfony.com/2.3/Symfony/Component/Form/Extension/Validator/ValidatorExtension.html

```
    $vendorFormDir.'/Resources/translations/validators.en.xlf',
    'en',
    'validators'
);
$translator->addResource(
    'xlf',
    $vendorValidatorDir.'/Resources/translations/validators.en.xlf',
    'en',
    'validators'
);

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->addExtension(new ValidatorExtension($validator))
    ->getFormFactory();
```

To learn more, skip down to the *Form Validation* section.

Accessing the Form Factory

Your application only needs one form factory, and that one factory object should be used to create any and all form objects in your application. This means that you should create it in some central, bootstrap part of your application and then access it whenever you need to build a form.

In this document, the form factory is always a local variable called `$formFactory`. The point here is that you will probably need to create this object in some more "global" way so you can access it from anywhere.

Exactly how you gain access to your one form factory is up to you. If you're using a *Service Container*, then you should add the form factory to your container and grab it out whenever you need to. If your application uses global or static variables (not usually a good idea), then you can store the object on some static class or do something similar.

Regardless of how you architect your application, just remember that you should only have one form factory and that you'll need to be able to access it throughout your application.

Creating a simple Form

If you're using the Symfony framework, then the form factory is available automatically as a service called `form.factory`. Also, the default base controller class has a *`createFormBuilder()`*[19] method, which is a shortcut to fetch the form factory and call `createBuilder` on it.

Creating a form is done via a *`FormBuilder`*[20] object, where you build and configure different fields. The form builder is created from the form factory.

Listing 43-13
```
$form = $formFactory->createBuilder()
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->getForm();
```

19. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Controller.html#createFormBuilder()
20. http://api.symfony.com/2.3/Symfony/Component/Form/FormBuilder.html

```
5
6 echo $twig->render('new.html.twig', array(
7     'form' => $form->createView(),
8 ));
```

As you can see, creating a form is like writing a recipe: you call `add` for each new field you want to create. The first argument to `add` is the name of your field, and the second is the field "type". The Form component comes with a lot of *built-in types*.

Now that you've built your form, learn how to *render* it and *process the form submission*.

Setting default Values

If you need your form to load with some default values (or you're building an "edit" form), simply pass in the default data when creating your form builder:

Listing 43-14
```
1 $defaults = array(
2     'dueDate' => new \DateTime('tomorrow'),
3 );
4
5 $form = $formFactory->createBuilder('form', $defaults)
6     ->add('task', 'text')
7     ->add('dueDate', 'date')
8     ->getForm();
```

In this example, the default data is an array. Later, when you use the *data_class* option to bind data directly to objects, your default data will be an instance of that object.

Rendering the Form

Now that the form has been created, the next step is to render it. This is done by passing a special form "view" object to your template (notice the `$form->createView()` in the controller above) and using a set of form helper functions:

Listing 43-15
```
1 <form action="#" method="post" {{ form_enctype(form) }}>
2     {{ form_widget(form) }}
3
4     <input type="submit" />
5 </form>
```

That's it! By printing `form_widget(form)`, each field in the form is rendered, along with a label and error message (if there is one). As easy as this is, it's not very flexible (yet). Usually, you'll want to render

each form field individually so you can control how the form looks. You'll learn how to do that in the "*Rendering a Form in a Template*" section.

Changing a Form's Method and Action

New in version 2.3: The ability to configure the form method and action was introduced in Symfony 2.3.

By default, a form is submitted to the same URI that rendered the form with an HTTP POST request. This behavior can be changed using the *action* and *method* options (the `method` option is also used by `handleRequest()` to determine whether a form has been submitted):

Listing 43-16
```php
$formBuilder = $formFactory->createBuilder('form', null, array(
    'action' => '/search',
    'method' => 'GET',
));

// ...
```

Handling Form Submissions

To handle form submissions, use the *`handleRequest()`*[21] method:

Listing 43-17
```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RedirectResponse;

$form = $formFactory->createBuilder()
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->getForm();

$request = Request::createFromGlobals();

$form->handleRequest($request);

if ($form->isValid()) {
    $data = $form->getData();

    // ... perform some action, such as saving the data to the database

    $response = new RedirectResponse('/task/success');
    $response->prepare($request);

    return $response->send();
}

// ...
```

This defines a common form "workflow", which contains 3 different possibilities:

1. On the initial GET request (i.e. when the user "surfs" to your page), build your form and render it;

If the request is a POST, process the submitted data (via `handleRequest()`). Then:

2. if the form is invalid, re-render the form (which will now contain errors);
3. if the form is valid, perform some action and redirect.

Luckily, you don't need to decide whether or not a form has been submitted. Just pass the current request to the `handleRequest()` method. Then, the Form component will do all the necessary work for you.

21. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#handleRequest()

Form Validation

The easiest way to add validation to your form is via the `constraints` option when building each field:

Listing 43-18
```php
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Type;

$form = $formFactory->createBuilder()
    ->add('task', 'text', array(
        'constraints' => new NotBlank(),
    ))
    ->add('dueDate', 'date', array(
        'constraints' => array(
            new NotBlank(),
            new Type('\DateTime'),
        )
    ))
    ->getForm();
```

When the form is bound, these validation constraints will be applied automatically and the errors will display next to the fields on error.

For a list of all of the built-in validation constraints, see *Validation Constraints Reference*.

Accessing Form Errors

You can use the *`getErrors()`*[22] method to access the list of errors. Each element is a *`FormError`*[23] object:

Listing 43-19
```php
$form = ...;

// ...

// an array of FormError objects, but only errors attached to this
// form level (e.g. "global errors)
$errors = $form->getErrors();

// an array of FormError objects, but only errors attached to the
// "firstName" field
$errors = $form['firstName']->getErrors();

// a string representation of all errors of the whole form tree
$errors = $form->getErrorsAsString();
```

If you enable the *error_bubbling* option on a field, calling `getErrors()` on the parent form will include errors from that field. However, there is no way to determine which field an error was originally attached to.

22. http://api.symfony.com/2.3/Symfony/Component/Form/FormInterface.html#getErrors()
23. http://api.symfony.com/2.3/Symfony/Component/Form/FormError.html

Chapter 44

Creating a custom Type Guesser

The Form component can guess the type and some options of a form field by using type guessers. The component already includes a type guesser using the assertions of the Validation component, but you can also add your own custom type guessers.

Form Type Guessers in the Bridges

Symfony also provides some form type guessers in the bridges:

- *`PropelTypeGuesser`*[1] provided by the Propel1 bridge;
- *`DoctrineOrmTypeGuesser`*[2] provided by the Doctrine bridge.

Create a PHPDoc Type Guesser

In this section, you are going to build a guesser that reads information about fields from the PHPDoc of the properties. At first, you need to create a class which implements *`FormTypeGuesserInterface`*[3]. This interface requires 4 methods:

- *`guessType()`*[4] - tries to guess the type of a field;
- *`guessRequired()`*[5] - tries to guess the value of the *required* option;
- *`guessMaxLength()`*[6] - tries to guess the value of the *max_length* option;
- *`guessPattern()`*[7] - tries to guess the value of the *pattern* option.

Start by creating the class and these methods. Next, you'll learn how to fill each on.

Listing 44-1

1. http://api.symfony.com/2.3/Symfony/Bridge/Propel1/Form/PropelTypeGuesser.html
2. http://api.symfony.com/2.3/Symfony/Bridge/Doctrine/Form/DoctrineOrmTypeGuesser.html
3. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeGuesserInterface.html
4. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeGuesserInterface.html#guessType()
5. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeGuesserInterface.html#guessRequired()
6. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeGuesserInterface.html#guessMaxLength()
7. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeGuesserInterface.html#guessPattern()

```php
namespace Acme\Form;

use Symfony\Component\Form\FormTypeGuesserInterface;

class PHPDocTypeGuesser implements FormTypeGuesserInterface
{
    public function guessType($class, $property)
    {
    }

    public function guessRequired($class, $property)
    {
    }

    public function guessMaxLength($class, $property)
    {
    }

    public function guessPattern($class, $property)
    {
    }
}
```

Guessing the Type

When guessing a type, the method returns either an instance of *TypeGuess*[8] or nothing, to determine that the type guesser cannot guess the type.

The `TypeGuess` constructor requires 3 options:

- The type name (one of the *form types*);
- Additional options (for instance, when the type is `entity`, you also want to set the `class` option). If no types are guessed, this should be set to an empty array;
- The confidence that the guessed type is correct. This can be one of the constants of the *Guess*[9] class: `LOW_CONFIDENCE`, `MEDIUM_CONFIDENCE`, `HIGH_CONFIDENCE`, `VERY_HIGH_CONFIDENCE`. After all type guessers have been executed, the type with the highest confidence is used.

With this knowledge, you can easily implement the `guessType` method of the `PHPDocTypeGuesser`:

Listing 44-2
```php
namespace Acme\Form;

use Symfony\Component\Form\Guess\Guess;
use Symfony\Component\Form\Guess\TypeGuess;

class PHPDocTypeGuesser implements FormTypeGuesserInterface
{
    public function guessType($class, $property)
    {
        $annotations = $this->readPhpDocAnnotations($class, $property);

        if (!isset($annotations['var'])) {
            return; // guess nothing if the @var annotation is not available
        }

```

8. http://api.symfony.com/2.3/Symfony/Component/Form/Guess/TypeGuess.html
9. http://api.symfony.com/2.3/Symfony/Component/Form/Guess/Guess.html

```
        // otherwise, base the type on the @var annotation
        switch ($annotations['var']) {
            case 'string':
                // there is a high confidence that the type is text when
                // @var string is used
                return new TypeGuess('text', array(), Guess::HIGH_CONFIDENCE);

            case 'int':
            case 'integer':
                // integers can also be the id of an entity or a checkbox (0 or 1)
                return new TypeGuess('integer', array(), Guess::MEDIUM_CONFIDENCE);

            case 'float':
            case 'double':
            case 'real':
                return new TypeGuess('number', array(), Guess::MEDIUM_CONFIDENCE);

            case 'boolean':
            case 'bool':
                return new TypeGuess('checkbox', array(), Guess::HIGH_CONFIDENCE);

            default:
                // there is a very low confidence that this one is correct
                return new TypeGuess('text', array(), Guess::LOW_CONFIDENCE);
        }
    }

    protected function readPhpDocAnnotations($class, $property)
    {
        $reflectionProperty = new \ReflectionProperty($class, $property);
        $phpdoc = $reflectionProperty->getDocComment();

        // parse the $phpdoc into an array like:
        // array('type' => 'string', 'since' => '1.0')
        $phpdocTags = ...;

        return $phpdocTags;
    }
}
```

This type guesser can now guess the field type for a property if it has PHPdoc!

Guessing Field Options

The other 3 methods (`guessMaxLength`, `guessRequired` and `guessPattern`) return a *`ValueGuess`*[10] instance with the value of the option. This constructor has 2 arguments:

- The value of the option;
- The confidence that the guessed value is correct (using the constants of the `Guess` class).

`null` is guessed when you believe the value of the option should not be set.

10. http://api.symfony.com/2.3/Symfony/Component/Form/Guess/ValueGuess.html

You should be very careful using the `guessPattern` method. When the type is a float, you cannot use it to determine a min or max value of the float (e.g. you want a float to be greater than `5`, `4.512313` is not valid but `length(4.512314) > length(5)` is, so the pattern will succeed). In this case, the value should be set to `null` with a `MEDIUM_CONFIDENCE`.

Registering a Type Guesser

The last thing you need to do is registering your custom type guesser by using *`addTypeGuesser()`*[11] or *`addTypeGuessers()`*[12]:

Listing 44-3
```php
use Symfony\Component\Form\Forms;
use Acme\Form\PHPDocTypeGuesser;

$formFactory = Forms::createFormFactoryBuilder()
    // ...
    ->addTypeGuesser(new PHPDocTypeGuesser())
    ->getFormFactory();

// ...
```

When you use the Symfony framework, you need to register your type guesser and tag it with `form.type_guesser`. For more information see *the tag reference*.

11. http://api.symfony.com/2.3/Symfony/Component/Form/FormFactoryBuilder.html#addTypeGuesser()
12. http://api.symfony.com/2.3/Symfony/Component/Form/FormFactoryBuilder.html#addTypeGuessers()

Chapter 45
Form Events

The Form component provides a structured process to let you customize your forms, by making use of the *EventDispatcher* component. Using form events, you may modify information or fields at different steps of the workflow: from the population of the form to the submission of the data from the request.

Registering an event listener is very easy using the Form component.

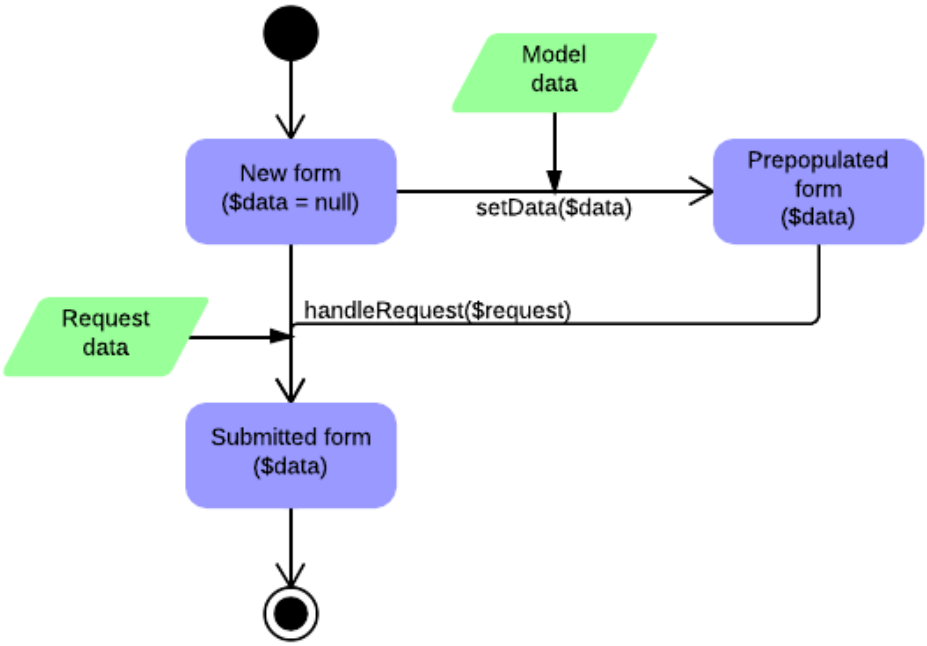
For example, if you wish to register a function to the `FormEvents::PRE_SUBMIT` event, the following code lets you add a field, depending on the request values:

Listing 45-1

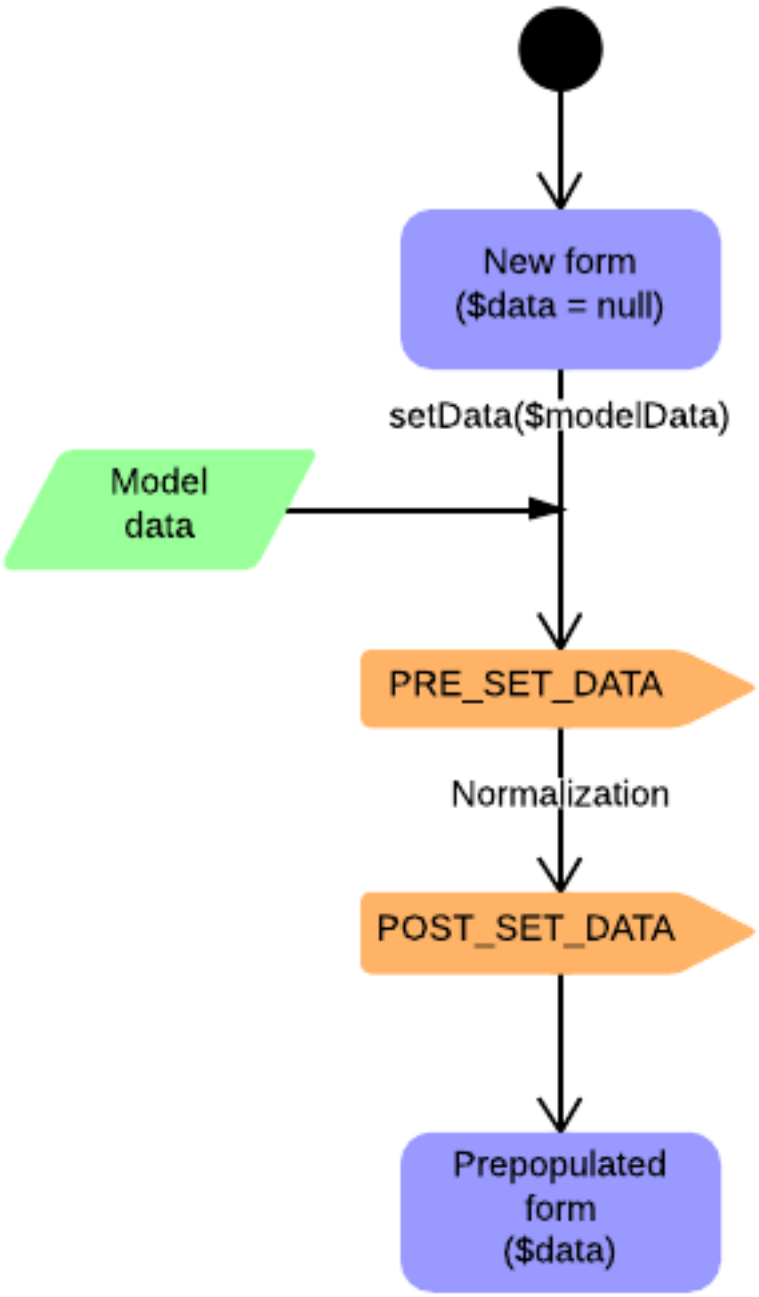
```
// ...

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;

$listener = function (FormEvent $event) {
    // ...
};

$form = $formFactory->createBuilder()
    // add form fields
    ->addEventListener(FormEvents::PRE_SUBMIT, $listener);

// ...
```

The Form Workflow

The Form Submission Workflow



1) Pre-populating the Form (`FormEvents::PRE_SET_DATA` and `FormEvents::POST_SET_DATA`)



Two events are dispatched during pre-population of a form, when *`Form::setData()`*[1] is called: `FormEvents::PRE_SET_DATA` and `FormEvents::POST_SET_DATA`.

A) The `FormEvents::PRE_SET_DATA` Event

The `FormEvents::PRE_SET_DATA` event is dispatched at the beginning of the `Form::setData()` method. It can be used to:

1. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#setData()

- Modify the data given during pre-population;
- Modify a form depending on the pre-populated data (adding or removing fields dynamically).

Form Events Information Table

Data Type	Value
Model data	`null`
Normalized data	`null`
View data	`null`

During `FormEvents::PRE_SET_DATA`, *`Form::setData()`*[2] is locked and will throw an exception if used. If you wish to modify data, you should use *`FormEvent::setData()`*[3] instead.

`FormEvents::PRE_SET_DATA` in the Form component

The `collection` form type relies on the `Symfony\Component\Form\Extension\Core\EventListener\ResizeFormListener` subscriber, listening to the `FormEvents::PRE_SET_DATA` event in order to reorder the form's fields depending on the data from the pre-populated object, by removing and adding all form rows.

B) The `FormEvents::POST_SET_DATA` Event

The `FormEvents::POST_SET_DATA` event is dispatched at the end of the *`Form::setData()`*[4] method. This event is mostly here for reading data after having pre-populated the form.

Form Events Information Table

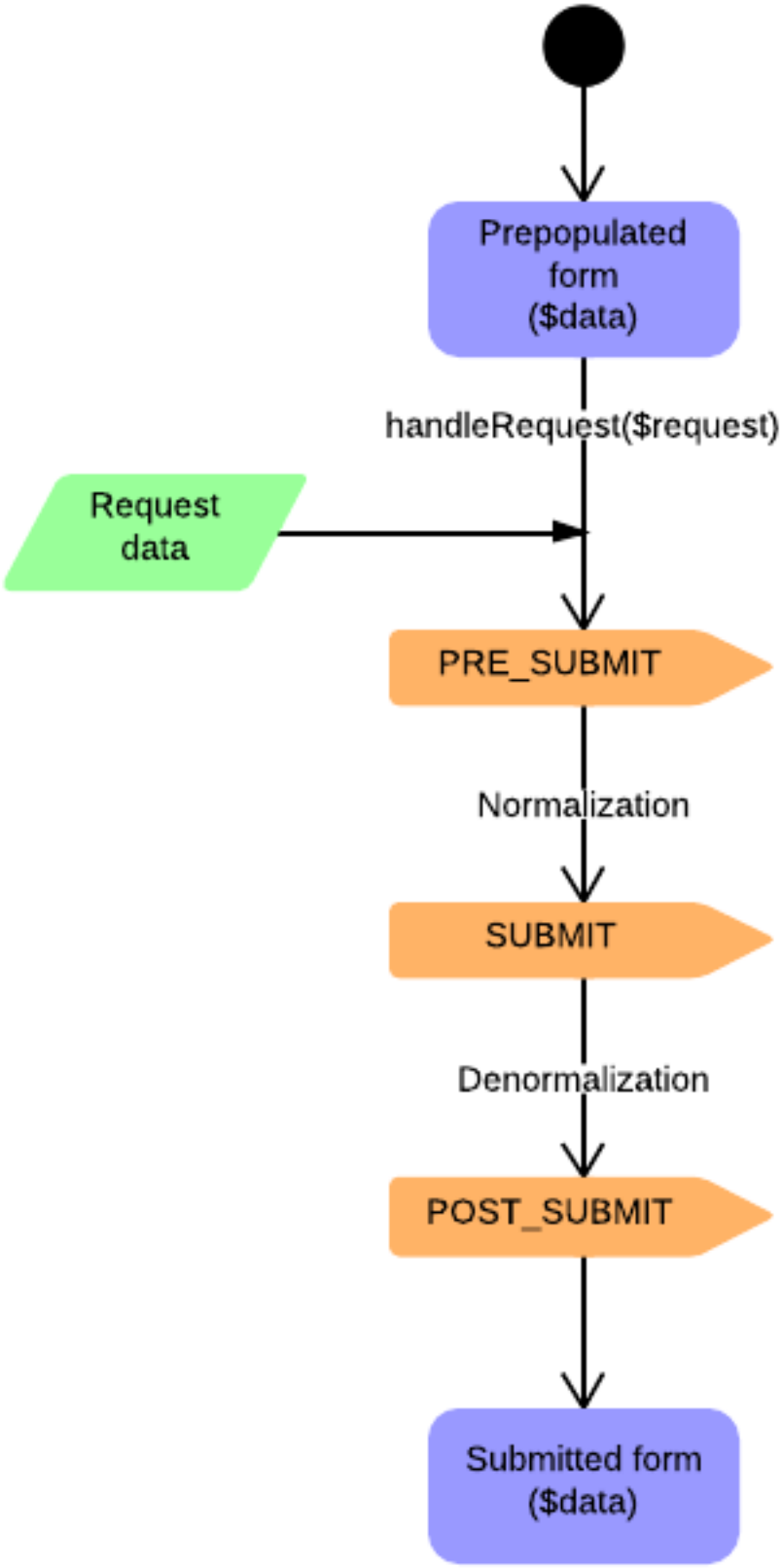
Data Type	Value
Model data	Model data injected into `setData()`
Normalized data	Model data transformed using a model transformer
View data	Normalized data transformed using a view transformer

`FormEvents::POST_SET_DATA` in the Form component

The `Symfony\Component\Form\Extension\DataCollector\EventListener\DataCollectorListener` class is subscribed to listen to the `FormEvents::POST_SET_DATA` event in order to collect information about the forms from the denormalized model and view data.

2. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#setData()
3. http://api.symfony.com/2.3/Symfony/Component/Form/FormEvent.html#setData()
4. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#setData()

2) Submitting a Form (`FormEvents::PRE_SUBMIT`, `FormEvents::SUBMIT` and `FormEvents::POST_SUBMIT`)


Prepopulated form ($data)
handleRequest($request)
Request data
PRE_SUBMIT
Normalization
SUBMIT
Denormalization
POST_SUBMIT
Submitted form ($data)

Three events are dispatched when *`Form::handleRequest()`*[5] or *`Form::submit()`*[6] are called: `FormEvents::PRE_SUBMIT`, `FormEvents::SUBMIT`, `FormEvents::POST_SUBMIT`.

A) The `FormEvents::PRE_SUBMIT` Event

The `FormEvents::PRE_SUBMIT` event is dispatched at the beginning of the *`Form::submit()`*[7] method.

It can be used to:

- Change data from the request, before submitting the data to the form;
- Add or remove form fields, before submitting the data to the form.

Form Events Information Table

Data Type	Value
Model data	Same as in `FormEvents::POST_SET_DATA`
Normalized data	Same as in `FormEvents::POST_SET_DATA`
View data	Same as in `FormEvents::POST_SET_DATA`

`FormEvents::PRE_SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\Core\EventListener\TrimListener` subscriber subscribes to the `FormEvents::PRE_SUBMIT` event in order to trim the request's data (for string values). The `Symfony\Component\Form\Extension\Csrf\EventListener\CsrfValidationListener` subscriber subscribes to the `FormEvents::PRE_SUBMIT` event in order to validate the CSRF token.

B) The `FormEvents::SUBMIT` Event

The `FormEvents::SUBMIT` event is dispatched just before the *`Form::submit()`*[8] method transforms back the normalized data to the model and view data.

It can be used to change data from the normalized representation of the data.

Form Events Information Table

Data Type	Value
Model data	Same as in `FormEvents::POST_SET_DATA`
Normalized data	Data from the request reverse-transformed from the request using a view transformer
View data	Same as in `FormEvents::POST_SET_DATA`

At this point, you cannot add or remove fields to the form.

5. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#handleRequest()
6. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#submit()
7. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#submit()
8. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#submit()

`FormEvents::SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\Core\EventListener\ResizeFormListener` subscribes to the `FormEvents::SUBMIT` event in order to remove the fields that need to be removed whenever manipulating a collection of forms for which `allow_delete` has been enabled.

C) The `FormEvents::POST_SUBMIT` Event

The `FormEvents::POST_SUBMIT` event is dispatched after the *`Form::submit()`*[9] once the model and view data have been denormalized.

It can be used to fetch data after denormalization.

Form Events Information Table

Data Type	Value
Model data	Normalized data reverse-transformed using a model transformer
Normalized data	Same as in `FormEvents::POST_SUBMIT`
View data	Normalized data transformed using a view transformer

At this point, you cannot add or remove fields to the form.

`FormEvents::POST_SUBMIT` in the Form component

The `Symfony\Component\Form\Extension\DataCollector\EventListener\DataCollectorListener` subscribes to the `FormEvents::POST_SUBMIT` event in order to collect information about the forms. The `Symfony\Component\Form\Extension\Validator\EventListener\ValidationListener` subscribes to the `FormEvents::POST_SUBMIT` event in order to automatically validate the denormalized object, and update the normalized as well as the view's representations.

Registering Event Listeners or Event Subscribers

In order to be able to use Form events, you need to create an event listener or an event subscriber, and register it to an event.

The name of each of the "form" events is defined as a constant on the *`FormEvents`*[10] class. Additionally, each event callback (listener or subscriber method) is passed a single argument, which is an instance of *`FormEvent`*[11]. The event object contains a reference to the current state of the form, and the current data being processed.

Name	`FormEvents` Constant	Event's Data
`form.pre_set_data`	`FormEvents::PRE_SET_DATA`	Model data

9. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html#submit()
10. http://api.symfony.com/2.3/Symfony/Component/Form/FormEvents.html
11. http://api.symfony.com/2.3/Symfony/Component/Form/FormEvent.html

Name	FormEvents Constant	Event's Data
form.post_set_data	FormEvents::POST_SET_DATA	Model data
form.pre_bind	FormEvents::PRE_SUBMIT	Request data
form.bind	FormEvents::SUBMIT	Normalized data
form.post_bind	FormEvents::POST_SUBMIT	View data

New in version 2.3: Before Symfony 2.3, `FormEvents::PRE_SUBMIT`, `FormEvents::SUBMIT` and `FormEvents::POST_SUBMIT` were called `FormEvents::PRE_BIND`, `FormEvents::BIND` and `FormEvents::POST_BIND`.

The `FormEvents::PRE_BIND`, `FormEvents::BIND` and `FormEvents::POST_BIND` constants will be removed in version 3.0 of Symfony. The event names still keep their original values, so make sure you use the `FormEvents` constants in your code for forward compatibility.

Event Listeners

An event listener may be any type of valid callable.

Creating and binding an event listener to the form is very easy:

Listing 45-2

```php
// ...

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;

$form = $formFactory->createBuilder()
    ->add('username', 'text')
    ->add('show_email', 'checkbox')
    ->addEventListener(FormEvents::PRE_SUBMIT, function (FormEvent $event) {
        $user = $event->getData();
        $form = $event->getForm();

        if (!$user) {
            return;
        }

        // Check whether the user has chosen to display his email or not.
        // If the data was submitted previously, the additional value that is
        // included in the request variables needs to be removed.
        if (true === $user['show_email']) {
            $form->add('email', 'email');
        } else {
            unset($user['email']);
            $event->setData($user);
        }
    })
    ->getForm();

// ...
```

When you have created a form type class, you can use one of its methods as a callback for better readability:

Listing 45-3

```php
// ...

class SubscriptionType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('username', 'text');
        $builder->add('show_email', 'checkbox');
        $builder->addEventListener(
            FormEvents::PRE_SET_DATA,
            array($this, 'onPreSetData')
        );
    }

    public function onPreSetData(FormEvent $event)
    {
        // ...
    }
}
```

Event Subscribers

Event subscribers have different uses:

- Improving readability;
- Listening to multiple events;
- Regrouping multiple listeners inside a single class.

Listing 45-4

```php
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;

class AddEmailFieldListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            FormEvents::PRE_SET_DATA => 'onPreSetData',
            FormEvents::PRE_SUBMIT   => 'onPreSubmit',
        );
    }

    public function onPreSetData(FormEvent $event)
    {
        $user = $event->getData();
        $form = $event->getForm();

        // Check whether the user from the initial data has chosen to
        // display his email or not.
        if (true === $user->isShowEmail()) {
            $form->add('email', 'email');
        }
    }

    public function onPreSubmit(FormEvent $event)
    {
```

```
        $user = $event->getData();
        $form = $event->getForm();

        if (!$user) {
            return;
        }

        // Check whether the user has chosen to display his email or not.
        // If the data was submitted previously, the additional value that
        // is included in the request variables needs to be removed.
        if (true === $user['show_email']) {
            $form->add('email', 'email');
        } else {
            unset($user['email']);
            $event->setData($user);
        }
    }
}
```

To register the event subscriber, use the addEventSubscriber() method:

Listing 45-5

```
// ...

$form = $formFactory->createBuilder()
    ->add('username', 'text')
    ->add('show_email', 'checkbox')
    ->addEventSubscriber(new AddEmailFieldListener())
    ->getForm();

// ...
```

Chapter 46
The HttpFoundation Component

> The HttpFoundation component defines an object-oriented layer for the HTTP specification.

In PHP, the request is represented by some global variables (`$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION`, ...) and the response is generated by some functions (`echo`, `header`, `setcookie`, ...).

The Symfony HttpFoundation component replaces these default PHP global variables and functions by an object-oriented layer.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/http-foundation` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/HttpFoundation*[2]).

Request

The most common way to create a request is to base it on the current PHP global variables with *`createFromGlobals()`*[3]:

Listing 46-1
```php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

which is almost equivalent to the more verbose, but also more flexible, *`__construct()`*[4] call:

1. https://packagist.org/packages/symfony/http-foundation
2. https://github.com/symfony/HttpFoundation
3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#createFromGlobals()
4. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#__construct()

Listing 46-2
```php
$request = new Request(
    $_GET,
    $_POST,
    array(),
    $_COOKIE,
    $_FILES,
    $_SERVER
);
```

Accessing Request Data

A Request object holds information about the client request. This information can be accessed via several public properties:

- `request`: equivalent of `$_POST`;
- `query`: equivalent of `$_GET` (`$request->query->get('name')`);
- `cookies`: equivalent of `$_COOKIE`;
- `attributes`: no equivalent - used by your app to store other data (see *below*);
- `files`: equivalent of `$_FILES`;
- `server`: equivalent of `$_SERVER`;
- `headers`: mostly equivalent to a sub-set of `$_SERVER` (`$request->headers->get('User-Agent')`).

Each property is a *ParameterBag*[5] instance (or a sub-class of), which is a data holder class:

- `request`: *ParameterBag*[6];
- `query`: *ParameterBag*[7];
- `cookies`: *ParameterBag*[8];
- `attributes`: *ParameterBag*[9];
- `files`: *FileBag*[10];
- `server`: *ServerBag*[11];
- `headers`: *HeaderBag*[12].

All *ParameterBag*[13] instances have methods to retrieve and update its data:

all()[14]
: Returns the parameters.

keys()[15]
: Returns the parameter keys.

replace()[16]
: Replaces the current parameters by a new set.

5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
6. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
7. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
8. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
9. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
10. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/FileBag.html
11. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ServerBag.html
12. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/HeaderBag.html
13. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
14. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#all()**
15. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#keys()**
16. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#replace()**

add()[17]
: Adds parameters.

get()[18]
: Returns a parameter by name.

set()[19]
: Sets a parameter by name.

has()[20]
: Returns `true` if the parameter is defined.

remove()[21]
: Removes a parameter.

The *ParameterBag*[22] instance also has some methods to filter the input values:

getAlpha()[23]
: Returns the alphabetic characters of the parameter value;

getAlnum()[24]
: Returns the alphabetic characters and digits of the parameter value;

getDigits()[25]
: Returns the digits of the parameter value;

getInt()[26]
: Returns the parameter value converted to integer;

filter()[27]
: Filters the parameter by using the PHP *filter_var*[28] function.

All getters takes up to three arguments: the first one is the parameter name and the second one is the default value to return if the parameter does not exist:

Listing 46-3
```php
// the query string is '?foo=bar'

$request->query->get('foo');
// returns bar

$request->query->get('bar');
// returns null

$request->query->get('bar', 'bar');
// returns 'bar'
```

17. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#add()**
18. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#get()**
19. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#set()**
20. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#has()**
21. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#remove()**
22. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
23. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#getAlpha()**
24. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#getAlnum()**
25. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#getDigits()**
26. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#getInt()**
27. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html#filter()**
28. http://php.net/manual/en/function.filter-var.php

When PHP imports the request query, it handles request parameters like `foo[bar]=bar` in a special way as it creates an array. So you can get the `foo` parameter and you will get back an array with a `bar` element. But sometimes, you might want to get the value for the "original" parameter name: `foo[bar]`. This is possible with all the `ParameterBag` getters like *`get()`*[29] via the third argument:

Listing 46-4
```php
// the query string is '?foo[bar]=bar'

$request->query->get('foo');
// returns array('bar' => 'bar')

$request->query->get('foo[bar]');
// returns null

$request->query->get('foo[bar]', null, true);
// returns 'bar'
```

Thanks to the public `attributes` property, you can store additional data in the request, which is also an instance of *`ParameterBag`*[30]. This is mostly used to attach information that belongs to the Request and that needs to be accessed from many different points in your application. For information on how this is used in the Symfony framework, see *the Symfony book*.

Finally, the raw data sent with the request body can be accessed using *`getContent()`*[31]:

Listing 46-5
```php
$content = $request->getContent();
```

For instance, this may be useful to process a JSON string sent to the application by a remote service using the HTTP POST method.

Identifying a Request

In your application, you need a way to identify a request; most of the time, this is done via the "path info" of the request, which can be accessed via the *`getPathInfo()`*[32] method:

Listing 46-6
```php
// for a request to http://example.com/blog/index.php/post/hello-world
// the path info is "/post/hello-world"
$request->getPathInfo();
```

Simulating a Request

Instead of creating a request based on the PHP globals, you can also simulate a request:

Listing 46-7
```php
$request = Request::create(
    '/hello-world',
    'GET',
    array('name' => 'Fabien')
);
```

The *`create()`*[33] method creates a request based on a URI, a method and some parameters (the query parameters or the request ones depending on the HTTP method); and of course, you can also override all other variables as well (by default, Symfony creates sensible defaults for all the PHP global variables).

29. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#get()
30. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ParameterBag.html
31. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getContent()
32. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getPathInfo()
33. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#create()

Based on such a request, you can override the PHP global variables via *overrideGlobals()*[34]:

Listing 46-8
```
$request->overrideGlobals();
```

You can also duplicate an existing request via *duplicate()*[35] or change a bunch of parameters with a single call to *initialize()*[36].

Accessing the Session

If you have a session attached to the request, you can access it via the *getSession()*[37] method; the *hasPreviousSession()*[38] method tells you if the request contains a session which was started in one of the previous requests.

Accessing *Accept-** Headers Data

You can easily access basic data extracted from `Accept-*` headers by using the following methods:

getAcceptableContentTypes()[39]
: Returns the list of accepted content types ordered by descending quality.

getLanguages()[40]
: Returns the list of accepted languages ordered by descending quality.

getCharsets()[41]
: Returns the list of accepted charsets ordered by descending quality.

New in version 2.2: The *AcceptHeader*[42] class was introduced in Symfony 2.2.

If you need to get full access to parsed data from `Accept`, `Accept-Language`, `Accept-Charset` or `Accept-Encoding`, you can use *AcceptHeader*[43] utility class:

Listing 46-9
```
use Symfony\Component\HttpFoundation\AcceptHeader;

$accept = AcceptHeader::fromString($request->headers->get('Accept'));
if ($accept->has('text/html')) {
    $item = $accept->get('text/html');
    $charset = $item->getAttribute('charset', 'utf-8');
    $quality = $item->getQuality();
}

// Accept header items are sorted by descending quality
$accepts = AcceptHeader::fromString($request->headers->get('Accept'))
    ->all();
```

34. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#overrideGlobals()
35. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#duplicate()
36. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#initialize()
37. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getSession()
38. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#hasPreviousSession()
39. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getAcceptableContentTypes()**
40. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getLanguages()**
41. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getCharsets()**
42. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/AcceptHeader.html
43. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/AcceptHeader.html

Accessing other Data

The `Request` class has many other methods that you can use to access the request information. Have a look at *the `Request` API*[44] for more information about them.

Response

A *`Response`*[45] object holds all the information that needs to be sent back to the client from a given request. The constructor takes up to three arguments: the response content, the status code, and an array of HTTP headers:

Listing 46-10
```php
use Symfony\Component\HttpFoundation\Response;

$response = new Response(
    'Content',
    200,
    array('content-type' => 'text/html')
);
```

This information can also be manipulated after the Response object creation:

Listing 46-11
```php
$response->setContent('Hello World');

// the headers public attribute is a ResponseHeaderBag
$response->headers->set('Content-Type', 'text/plain');

$response->setStatusCode(404);
```

When setting the `Content-Type` of the Response, you can set the charset, but it is better to set it via the *`setCharset()`*[46] method:

Listing 46-12
```php
$response->setCharset('ISO-8859-1');
```

Note that by default, Symfony assumes that your Responses are encoded in UTF-8.

Sending the Response

Before sending the Response, you can ensure that it is compliant with the HTTP specification by calling the *`prepare()`*[47] method:

Listing 46-13
```php
$response->prepare($request);
```

Sending the response to the client is then as simple as calling *`send()`*[48]:

Listing 46-14
```php
$response->send();
```

44. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
45. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html
46. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setCharset()
47. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#prepare()
48. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#send()

Setting Cookies

The response cookies can be manipulated through the `headers` public attribute:

Listing 46-15
```
use Symfony\Component\HttpFoundation\Cookie;

$response->headers->setCookie(new Cookie('foo', 'bar'));
```

The *`setCookie()`*[49] method takes an instance of *`Cookie`*[50] as an argument.

You can clear a cookie via the *`clearCookie()`*[51] method.

Managing the HTTP Cache

The *`Response`*[52] class has a rich set of methods to manipulate the HTTP headers related to the cache:

- *`setPublic()`*[53];
- *`setPrivate()`*[54];
- *`expire()`*[55];
- *`setExpires()`*[56];
- *`setMaxAge()`*[57];
- *`setSharedMaxAge()`*[58];
- *`setTtl()`*[59];
- *`setClientTtl()`*[60];
- *`setLastModified()`*[61];
- *`setEtag()`*[62];
- *`setVary()`*[63];

The *`setCache()`*[64] method can be used to set the most commonly used cache information in one method call:

Listing 46-16
```
$response->setCache(array(
    'etag'          => 'abcdef',
    'last_modified' => new \DateTime(),
    'max_age'       => 600,
    's_maxage'      => 600,
    'private'       => false,
    'public'        => true,
));
```

To check if the Response validators (`ETag`, `Last-Modified`) match a conditional value specified in the client Request, use the *`isNotModified()`*[65] method:

---

49. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#setCookie()
50. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Cookie.html
51. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#clearCookie()
52. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html
53. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setPublic()
54. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setPrivate()
55. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#expire()
56. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setExpires()
57. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setMaxAge()
58. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setSharedMaxAge()
59. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setTtl()
60. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setClientTtl()
61. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setLastModified()
62. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setEtag()
63. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setVary()
64. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#setCache()

Listing 46-17
```php
if ($response->isNotModified($request)) {
    $response->send();
}
```

If the Response is not modified, it sets the status code to 304 and removes the actual response content.

Redirecting the User

To redirect the client to another URL, you can use the *RedirectResponse*[66] class:

Listing 46-18
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://example.com/');
```

Streaming a Response

The *StreamedResponse*[67] class allows you to stream the Response back to the client. The response content is represented by a PHP callable instead of a string:

Listing 46-19
```php
use Symfony\Component\HttpFoundation\StreamedResponse;

$response = new StreamedResponse();
$response->setCallback(function () {
    echo 'Hello World';
    flush();
    sleep(2);
    echo 'Hello World';
    flush();
});
$response->send();
```

The `flush()` function does not flush buffering. If `ob_start()` has been called before or the `output_buffering` `php.ini` option is enabled, you must call `ob_flush()` before `flush()`.

Additionally, PHP isn't the only layer that can buffer output. Your web server might also buffer based on its configuration. Even more, if you use fastcgi, buffering can't be disabled at all.

Serving Files

When sending a file, you must add a `Content-Disposition` header to your response. While creating this header for basic file downloads is easy, using non-ASCII filenames is more involving. The *makeDisposition()*[68] abstracts the hard work behind a simple API:

Listing 46-20
```php
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

$d = $response->headers->makeDisposition(
    ResponseHeaderBag::DISPOSITION_ATTACHMENT,
```

65. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#isNotModified()
66. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/RedirectResponse.html
67. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/StreamedResponse.html
68. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#makeDisposition()

```
    'foo.pdf'
);

$response->headers->set('Content-Disposition', $d);
```

New in version 2.2: The *BinaryFileResponse*[69] class was introduced in Symfony 2.2.

Alternatively, if you are serving a static file, you can use a *BinaryFileResponse*[70]:

Listing 46-21
```
use Symfony\Component\HttpFoundation\BinaryFileResponse;

$file = 'path/to/file.txt';
$response = new BinaryFileResponse($file);
```

The `BinaryFileResponse` will automatically handle `Range` and `If-Range` headers from the request. It also supports `X-Sendfile` (see for *Nginx*[71] and *Apache*[72]). To make use of it, you need to determine whether or not the `X-Sendfile-Type` header should be trusted and call *trustXSendfileTypeHeader()*[73] if it should:

Listing 46-22
```
BinaryFileResponse::trustXSendfileTypeHeader();
```

You can still set the `Content-Type` of the sent file, or change its `Content-Disposition`:

Listing 46-23
```
$response->headers->set('Content-Type', 'text/plain');
$response->setContentDisposition(
    ResponseHeaderBag::DISPOSITION_ATTACHMENT,
    'filename.txt'
);
```

Creating a JSON Response

Any type of response can be created via the *Response*[74] class by setting the right content and headers. A JSON response might look like this:

Listing 46-24
```
use Symfony\Component\HttpFoundation\Response;

$response = new Response();
$response->setContent(json_encode(array(
    'data' => 123,
)));
$response->headers->set('Content-Type', 'application/json');
```

There is also a helpful *JsonResponse*[75] class, which can make this even easier:

Listing 46-25
```
use Symfony\Component\HttpFoundation\JsonResponse;

```

69. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/BinaryFileResponse.html
70. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/BinaryFileResponse.html
71. http://wiki.nginx.org/XSendfile
72. https://tn123.org/mod_xsendfile/
73. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/BinaryFileResponse.html#trustXSendfileTypeHeader()
74. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html
75. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/JsonResponse.html

```
3 $response = new JsonResponse();
4 $response->setData(array(
5     'data' => 123
6 ));
```

This encodes your array of data to JSON and sets the `Content-Type` header to `application/json`.

To avoid XSSI *JSON Hijacking*[76], you should pass an associative array as the outer-most array to `JsonResponse` and not an indexed array so that the final result is an object (e.g. `{"object": "not inside an array"}`) instead of an array (e.g. `[{"object": "inside an array"}]`). Read the *OWASP guidelines*[77] for more information.

Only methods that respond to GET requests are vulnerable to XSSI 'JSON Hijacking'. Methods responding to POST requests only remain unaffected.

JSONP Callback

If you're using JSONP, you can set the callback function that the data should be passed to:

Listing 46-26
```
1 $response->setCallback('handleResponse');
```

In this case, the `Content-Type` header will be `text/javascript` and the response content will look like this:

Listing 46-27
```
1 handleResponse({'data': 123});
```

Session

The session information is in its own document: *Session Management*.

76. http://haacked.com/archive/2009/06/25/json-hijacking.aspx
77. https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines#Always_return_JSON_with_an_Object_on_the_outside

Chapter 47
Session Management

The Symfony HttpFoundation component has a very powerful and flexible session subsystem which is designed to provide session management through a simple object-oriented interface using a variety of session storage drivers.

Sessions are used via the simple *`Session`*[1] implementation of *`SessionInterface`*[2] interface.

Make sure your PHP session isn't already started before using the Session class. If you have a legacy session system that starts your session, see *Legacy Sessions*.

Quick example:

Listing 47-1

```php
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
$session->start();

// set and get session attributes
$session->set('name', 'Drak');
$session->get('name');

// set flash messages
$session->getFlashBag()->add('notice', 'Profile updated');

// retrieve messages
foreach ($session->getFlashBag()->get('notice', array()) as $message) {
    echo '<div class="flash-notice">'.$message.'</div>';
}
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionInterface.html

Symfony sessions are designed to replace several native PHP functions. Applications should avoid using `session_start()`, `session_regenerate_id()`, `session_id()`, `session_name()`, and `session_destroy()` and instead use the APIs in the following section.

While it is recommended to explicitly start a session, a session will actually start on demand, that is, if any session request is made to read/write session data.

Symfony sessions are incompatible with `php.ini` directive `session.auto_start = 1` This directive should be turned off in `php.ini`, in the webserver directives or in `.htaccess`.

Session API

The *`Session`*[3] class implements *`SessionInterface`*[4].

The *`Session`*[5] has a simple API as follows divided into a couple of groups.

Session Workflow

`start()`[6]
: Starts the session - do not use `session_start()`.

`migrate()`[7]
: Regenerates the session ID - do not use `session_regenerate_id()`. This method can optionally change the lifetime of the new cookie that will be emitted by calling this method.

`invalidate()`[8]
: Clears all session data and regenerates session ID. Do not use `session_destroy()`.

`getId()`[9]
: Gets the session ID. Do not use `session_id()`.

`setId()`[10]
: Sets the session ID. Do not use `session_id()`.

`getName()`[11]
: Gets the session name. Do not use `session_name()`.

`setName()`[12]
: Sets the session name. Do not use `session_name()`.

---

3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html
4. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionInterface.html
5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html
6. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#start()**
7. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#migrate()**
8. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#invalidate()**
9. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getId()**
10. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#setId()**
11. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getName()**
12. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#setName()**

Session Attributes

set()[13]
: Sets an attribute by key.

get()[14]
: Gets an attribute by key.

all()[15]
: Gets all attributes as an array of key => value.

has()[16]
: Returns true if the attribute exists.

replace()[17]
: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.

remove()[18]
: Deletes an attribute by key.

clear()[19]
: Clear all attributes.

The attributes are stored internally in a "Bag", a PHP object that acts like an array. A few methods exist for "Bag" management:

registerBag()[20]
: Registers a *SessionBagInterface*[21].

getBag()[22]
: Gets a *SessionBagInterface*[23] by bag name.

getFlashBag()[24]
: Gets the *FlashBagInterface*[25]. This is just a shortcut for convenience.

Session Metadata

getMetadataBag()[26]
: Gets the *MetadataBag*[27] which contains information about the session.

13. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#set()
14. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#get()
15. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#all()
16. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#has()
17. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#replace()
18. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#remove()
19. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#clear()
20. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#registerBag()
21. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
22. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getBag()
23. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
24. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getFlashBag()
25. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
26. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getMetadataBag()
27. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/MetadataBag.html

Session Data Management

PHP's session management requires the use of the `$_SESSION` super-global, however, this interferes somewhat with code testability and encapsulation in an OOP paradigm. To help overcome this, Symfony uses *session bags* linked to the session to encapsulate a specific dataset of attributes or flash messages.

This approach also mitigates namespace pollution within the `$_SESSION` super-global because each bag stores all its data under a unique namespace. This allows Symfony to peacefully co-exist with other applications or libraries that might use the `$_SESSION` super-global and all data remains completely compatible with Symfony's session management.

Symfony provides two kinds of storage bags, with two separate implementations. Everything is written against interfaces so you may extend or create your own bag types if necessary.

`SessionBagInterface`[28] has the following API which is intended mainly for internal purposes:

`getStorageKey()`[29]
: Returns the key which the bag will ultimately store its array under in `$_SESSION`. Generally this value can be left at its default and is for internal use.

`initialize()`[30]
: This is called internally by Symfony session storage classes to link bag data to the session.

`getName()`[31]
: Returns the name of the session bag.

Attributes

The purpose of the bags implementing the *`AttributeBagInterface`*[32] is to handle session attribute storage. This might include things like user ID, and remember me login settings or other user based state information.

`AttributeBag`[33]
: This is the standard default implementation.

`NamespacedAttributeBag`[34]
: This implementation allows for attributes to be stored in a structured namespace.

Any plain key-value storage system is limited in the extent to which complex data can be stored since each key must be unique. You can achieve namespacing by introducing a naming convention to the keys so different parts of your application could operate without clashing. For example, `module1.foo` and `module2.foo`. However, sometimes this is not very practical when the attributes data is an array, for example a set of tokens. In this case, managing the array becomes a burden because you have to retrieve the array then process it and store it again:

Listing 47-2
```php
$tokens = array(
    'tokens' => array(
        'a' => 'a6c1e0b6',
        'b' => 'f4a7b1f3',
```

28. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
29. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#getStorageKey()**
30. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#initialize()**
31. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#getName()**
32. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
33. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBag.html**
34. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/NamespacedAttributeBag.html**

```
    ),
);
```

So any processing of this might quickly get ugly, even simply adding a token to the array:

Listing 47-3
```php
$tokens = $session->get('tokens');
$tokens['c'] = $value;
$session->set('tokens', $tokens);
```

With structured namespacing, the key can be translated to the array structure like this using a namespace character (defaults to `/`):

Listing 47-4
```php
$session->set('tokens/c', $value);
```

This way you can easily access a key within the stored array directly and easily.

AttributeBagInterface[35] has a simple API

set()[36]
: Sets an attribute by key.

get()[37]
: Gets an attribute by key.

all()[38]
: Gets all attributes as an array of key => value.

has()[39]
: Returns true if the attribute exists.

keys()[40]
: Returns an array of stored attribute keys.

replace()[41]
: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.

remove()[42]
: Deletes an attribute by key.

clear()[43]
: Clear the bag.

35. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
36. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#set()
37. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#get()
38. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#all()
39. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#has()
40. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#keys()
41. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#replace()
42. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#remove()
43. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#clear()

Flash Messages

The purpose of the *FlashBagInterface*[44] is to provide a way of setting and retrieving messages on a per session basis. The usual workflow would be to set flash messages in a request and to display them after a page redirect. For example, a user submits a form which hits an update controller, and after processing the controller redirects the page to either the updated page or an error page. Flash messages set in the previous page request would be displayed immediately on the subsequent page load for that session. This is however just one application for flash messages.

AutoExpireFlashBag[45]
: In this implementation, messages set in one page-load will be available for display only on the next page load. These messages will auto expire regardless of if they are retrieved or not.

FlashBag[46]
: In this implementation, messages will remain in the session until they are explicitly retrieved or cleared. This makes it possible to use ESI caching.

FlashBagInterface[47] has a simple API

add()[48]
: Adds a flash message to the stack of specified type.

set()[49]
: Sets flashes by type; This method conveniently takes both single messages as a `string` or multiple messages in an `array`.

get()[50]
: Gets flashes by type and clears those flashes from the bag.

setAll()[51]
: Sets all flashes, accepts a keyed array of arrays `type => array(messages)`.

all()[52]
: Gets all flashes (as a keyed array of arrays) and clears the flashes from the bag.

peek()[53]
: Gets flashes by type (read only).

peekAll()[54]
: Gets all flashes (read only) as keyed array of arrays.

has()[55]
: Returns true if the type exists, false if not.

keys()[56]
: Returns an array of the stored flash types.

---

44. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
45. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/AutoExpireFlashBag.html**
46. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBag.html**
47. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
48. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#add()**
49. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#set()**
50. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#get()**
51. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#setAll()**
52. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#all()**
53. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#peek()**
54. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#peekAll()**
55. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#has()**
56. **http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#keys()**

clear()[57]
: Clears the bag.

For simple applications it is usually sufficient to have one flash message per type, for example a confirmation notice after a form is submitted. However, flash messages are stored in a keyed array by flash `$type` which means your application can issue multiple messages for a given type. This allows the API to be used for more complex messaging in your application.

Examples of setting multiple flashes:

Listing 47-5
```php
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
$session->start();

// add flash messages
$session->getFlashBag()->add(
    'warning',
    'Your config file is writable, it should be set read-only'
);
$session->getFlashBag()->add('error', 'Failed to update name');
$session->getFlashBag()->add('error', 'Another error');
```

Displaying the flash messages might look as follows.

Simple, display one type of message:

Listing 47-6
```php
// display warnings
foreach ($session->getFlashBag()->get('warning', array()) as $message) {
    echo '<div class="flash-warning">'.$message.'</div>';
}

// display errors
foreach ($session->getFlashBag()->get('error', array()) as $message) {
    echo '<div class="flash-error">'.$message.'</div>';
}
```

Compact method to process display all flashes at once:

Listing 47-7
```php
foreach ($session->getFlashBag()->all() as $type => $messages) {
    foreach ($messages as $message) {
        echo '<div class="flash-'.$type.'">'.$message.'</div>';
    }
}
```

57. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#clear()

Chapter 48
Configuring Sessions and Save Handlers

This section deals with how to configure session management and fine tune it to your specific needs. This documentation covers save handlers, which store and retrieve session data, and configuring session behavior.

Save Handlers

The PHP session workflow has 6 possible operations that may occur. The normal session follows `open`, `read`, `write` and `close`, with the possibility of `destroy` and `gc` (garbage collection which will expire any old sessions: `gc` is called randomly according to PHP's configuration and if called, it is invoked after the `open` operation). You can read more about this at *php.net/session.customhandler*[1]

Native PHP Save Handlers

So-called native handlers, are save handlers which are either compiled into PHP or provided by PHP extensions, such as PHP-Sqlite, PHP-Memcached and so on.

All native save handlers are internal to PHP and as such, have no public facing API. They must be configured by `php.ini` directives, usually `session.save_path` and potentially other driver specific directives. Specific details can be found in the docblock of the `setOptions()` method of each class. For instance, the one provided by the Memcached extension can be found on *php.net/memcached.setoption*[2]

While native save handlers can be activated by directly using `ini_set('session.save_handler', $name);`, Symfony provides a convenient way to activate these in the same way as it does for custom handlers.

Symfony provides drivers for the following native save handler as an example:

- *`NativeFileSessionHandler`*[3]

Example usage:

1. http://php.net/session.customhandler
2. http://php.net/memcached.setoption
3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeFileSessionHandler.html

Listing 48-1
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
use Symfony\Component\HttpFoundation\Session\Storage\Handler\NativeFileSessionHandler;

$storage = new NativeSessionStorage(array(), new NativeFileSessionHandler());
$session = new Session($storage);
```

With the exception of the `files` handler which is built into PHP and always available, the availability of the other handlers depends on those PHP extensions being active at runtime.

Native save handlers provide a quick solution to session storage, however, in complex systems where you need more control, custom save handlers may provide more freedom and flexibility. Symfony provides several implementations which you may further customize as required.

Custom Save Handlers

Custom handlers are those which completely replace PHP's built-in session save handlers by providing six callback functions which PHP calls internally at various points in the session workflow.

The Symfony HttpFoundation component provides some by default and these can easily serve as examples if you wish to write your own.

- *PdoSessionHandler*[4]
- *MemcacheSessionHandler*[5]
- *MemcachedSessionHandler*[6]
- *MongoDbSessionHandler*[7]
- *NullSessionHandler*[8]

Example usage:

Listing 48-2
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
use Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler;

$pdo = new \PDO(...);
$storage = new NativeSessionStorage(array(), new PdoSessionHandler($pdo));
$session = new Session($storage);
```

Configuring PHP Sessions

The *NativeSessionStorage*[9] can configure most of the `php.ini` configuration directives which are documented at *php.net/session.configuration*[10].

4. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html
5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcacheSessionHandler.html
6. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcachedSessionHandler.html
7. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/MongoDbSessionHandler.html
8. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/NullSessionHandler.html
9. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
10. http://php.net/session.configuration

To configure these settings, pass the keys (omitting the initial `session.` part of the key) as a key-value array to the `$options` constructor argument. Or set them via the *`setOptions()`*[11] method.

For the sake of clarity, some key options are explained in this documentation.

Session Cookie Lifetime

For security, session tokens are generally recommended to be sent as session cookies. You can configure the lifetime of session cookies by specifying the lifetime (in seconds) using the `cookie_lifetime` key in the constructor's `$options` argument in *`NativeSessionStorage`*[12].

Setting a `cookie_lifetime` to `0` will cause the cookie to live only as long as the browser remains open. Generally, `cookie_lifetime` would be set to a relatively large number of days, weeks or months. It is not uncommon to set cookies for a year or more depending on the application.

Since session cookies are just a client-side token, they are less important in controlling the fine details of your security settings which ultimately can only be securely controlled from the server side.

The `cookie_lifetime` setting is the number of seconds the cookie should live for, it is not a Unix timestamp. The resulting session cookie will be stamped with an expiry time of `time()` + `cookie_lifetime` where the time is taken from the server.

Configuring Garbage Collection

When a session opens, PHP will call the `gc` handler randomly according to the probability set by `session.gc_probability` / `session.gc_divisor`. For example if these were set to `5/100` respectively, it would mean a probability of 5%. Similarly, `3/4` would mean a 3 in 4 chance of being called, i.e. 75%.

If the garbage collection handler is invoked, PHP will pass the value stored in the `php.ini` directive `session.gc_maxlifetime`. The meaning in this context is that any stored session that was saved more than `gc_maxlifetime` ago should be deleted. This allows one to expire records based on idle time.

You can configure these settings by passing `gc_probability`, `gc_divisor` and `gc_maxlifetime` in an array to the constructor of *`NativeSessionStorage`*[13] or to the *`setOptions()`*[14] method.

Session Lifetime

When a new session is created, meaning Symfony issues a new session cookie to the client, the cookie will be stamped with an expiry time. This is calculated by adding the PHP runtime configuration value in `session.cookie_lifetime` with the current server time.

PHP will only issue a cookie once. The client is expected to store that cookie for the entire lifetime. A new cookie will only be issued when the session is destroyed, the browser cookie is deleted, or the session ID is regenerated using the `migrate()` or `invalidate()` methods of the `Session` class.

The initial cookie lifetime can be set by configuring `NativeSessionStorage` using the `setOptions(array('cookie_lifetime' => 1234))` method.

11. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#setOptions()
12. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
13. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
14. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#setOptions()

A cookie lifetime of `0` means the cookie expires when the browser is closed.

Session Idle Time/Keep Alive

There are often circumstances where you may want to protect, or minimize unauthorized use of a session when a user steps away from their terminal while logged in by destroying the session after a certain period of idle time. For example, it is common for banking applications to log the user out after just 5 to 10 minutes of inactivity. Setting the cookie lifetime here is not appropriate because that can be manipulated by the client, so we must do the expiry on the server side. The easiest way is to implement this via garbage collection which runs reasonably frequently. The `cookie_lifetime` would be set to a relatively high value, and the garbage collection `gc_maxlifetime` would be set to destroy sessions at whatever the desired idle period is.

The other option is specifically check if a session has expired after the session is started. The session can be destroyed as required. This method of processing can allow the expiry of sessions to be integrated into the user experience, for example, by displaying a message.

Symfony records some basic metadata about each session to give you complete freedom in this area.

Session Metadata

Sessions are decorated with some basic metadata to enable fine control over the security settings. The session object has a getter for the metadata, *`getMetadataBag()`*[15] which exposes an instance of *`MetadataBag`*[16]:

Listing 48-3
```
$session->getMetadataBag()->getCreated();
$session->getMetadataBag()->getLastUsed();
```

Both methods return a Unix timestamp (relative to the server).

This metadata can be used to explicitly expire a session on access, e.g.:

Listing 48-4
```
$session->start();
if (time() - $session->getMetadataBag()->getLastUsed() > $maxIdleTime) {
    $session->invalidate();
    throw new SessionExpired(); // redirect to expired session page
}
```

It is also possible to tell what the `cookie_lifetime` was set to for a particular cookie by reading the `getLifetime()` method:

Listing 48-5
```
$session->getMetadataBag()->getLifetime();
```

The expiry time of the cookie can be determined by adding the created timestamp and the lifetime.

15. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Session.html#getMetadataBag()
16. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/MetadataBag.html

PHP 5.4 Compatibility

Since PHP 5.4.0, *SessionHandler*[17] and *SessionHandlerInterface*[18] are available. Symfony provides forward compatibility for the *SessionHandlerInterface*[19] so it can be used under PHP 5.3. This greatly improves interoperability with other libraries.

SessionHandler[20] is a special PHP internal class which exposes native save handlers to PHP user-space.

In order to provide a solution for those using PHP 5.4, Symfony has a special class called *NativeSessionHandler*[21] which under PHP 5.4, extends from `\SessionHandler` and under PHP 5.3 is just a empty base class. This provides some interesting opportunities to leverage PHP 5.4 functionality if it is available.

Save Handler Proxy

A Save Handler Proxy is basically a wrapper around a Save Handler that was introduced to seamlessly support the migration from PHP 5.3 to PHP 5.4+. It further creates an extension point from where custom logic can be added that works independently of which handler is being wrapped inside.

There are two kinds of save handler class proxies which inherit from *AbstractProxy*[22]: they are *NativeProxy*[23] and *SessionHandlerProxy*[24].

NativeSessionStorage[25] automatically injects storage handlers into a save handler proxy unless already wrapped by one.

NativeProxy[26] is used automatically under PHP 5.3 when internal PHP save handlers are specified using the `Native*SessionHandler` classes, while *SessionHandlerProxy*[27] will be used to wrap any custom save handlers, that implement *SessionHandlerInterface*[28].

From PHP 5.4 and above, all session handlers implement *SessionHandlerInterface*[29] including `Native*SessionHandler` classes which inherit from *SessionHandler*[30].

The proxy mechanism allows you to get more deeply involved in session save handler classes. A proxy for example could be used to encrypt any session transaction without knowledge of the specific save handler.

Before PHP 5.4, you can only proxy user-land save handlers but not native PHP save handlers.

17. http://php.net/manual/en/class.sessionhandler.php
18. http://php.net/manual/en/class.sessionhandlerinterface.php
19. http://php.net/manual/en/class.sessionhandlerinterface.php
20. http://php.net/manual/en/class.sessionhandler.php
21. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeSessionHandler.html
22. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/AbstractProxy.html
23. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeProxy.html
24. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/SessionHandlerProxy.html
25. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
26. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeProxy.html
27. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/SessionHandlerProxy.html
28. http://php.net/manual/en/class.sessionhandlerinterface.php
29. http://php.net/manual/en/class.sessionhandlerinterface.php
30. http://php.net/manual/en/class.sessionhandler.php

Chapter 49
Testing with Sessions

Symfony is designed from the ground up with code-testability in mind. In order to make your code which utilizes session easily testable we provide two separate mock storage mechanisms for both unit testing and functional testing.

Testing code using real sessions is tricky because PHP's workflow state is global and it is not possible to have multiple concurrent sessions in the same PHP process.

The mock storage engines simulate the PHP session workflow without actually starting one allowing you to test your code without complications. You may also run multiple instances in the same PHP process.

The mock storage drivers do not read or write the system globals `session_id()` or `session_name()`. Methods are provided to simulate this if required:

- *`getId()`*[1]: Gets the session ID.
- *`setId()`*[2]: Sets the session ID.
- *`getName()`*[3]: Gets the session name.
- *`setName()`*[4]: Sets the session name.

Unit Testing

For unit testing where it is not necessary to persist the session, you should simply swap out the default storage engine with *`MockArraySessionStorage`*[5]:

Listing 49-1
```php
use Symfony\Component\HttpFoundation\Session\Storage\MockArraySessionStorage;
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session(new MockArraySessionStorage());
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#getId()
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#setId()
3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#getName()
4. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#setName()
5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/MockArraySessionStorage.html

Functional Testing

For functional testing where you may need to persist session data across separate PHP processes, simply change the storage engine to *MockFileSessionStorage*[6]:

Listing 49-2
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\MockFileSessionStorage;

$session = new Session(new MockFileSessionStorage());
```

6. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/MockFileSessionStorage.html

Chapter 50
Integrating with Legacy Sessions

Sometimes it may be necessary to integrate Symfony into a legacy application where you do not initially have the level of control you require.

As stated elsewhere, Symfony Sessions are designed to replace the use of PHP's native `session_*()` functions and use of the `$_SESSION` superglobal. Additionally, it is mandatory for Symfony to start the session.

However when there really are circumstances where this is not possible, you can use a special storage bridge *`PhpBridgeSessionStorage`*[1] which is designed to allow Symfony to work with a session started outside of the Symfony Session framework. You are warned that things can interrupt this use-case unless you are careful: for example the legacy application erases `$_SESSION`.

A typical use of this might look like this:

Listing 50-1
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\PhpBridgeSessionStorage;

// legacy application configures session
ini_set('session.save_handler', 'files');
ini_set('session.save_path', '/tmp');
session_start();

// Get Symfony to interface with this existing session
$session = new Session(new PhpBridgeSessionStorage());

// symfony will now interface with the existing PHP session
$session->start();
```

This will allow you to start using the Symfony Session API and allow migration of your application to Symfony sessions.

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/PhpBridgeSessionStorage.html

Symfony sessions store data like attributes in special 'Bags' which use a key in the `$_SESSION` superglobal. This means that a Symfony session cannot access arbitrary keys in `$_SESSION` that may be set by the legacy application, although all the `$_SESSION` contents will be saved when the session is saved.

Chapter 51
Trusting Proxies

If you're using the Symfony Framework, start by reading *How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy*.

If you find yourself behind some sort of proxy - like a load balancer - then certain header information may be sent to you using special `X-Forwarded-*` headers. For example, the `Host` HTTP header is usually used to return the requested host. But when you're behind a proxy, the true host may be stored in a `X-Forwarded-Host` header.

Since HTTP headers can be spoofed, Symfony does *not* trust these proxy headers by default. If you are behind a proxy, you should manually whitelist your proxy.

New in version 2.3: CIDR notation support was introduced in Symfony 2.3, so you can whitelist whole subnets (e.g. `10.0.0.0/8`, `fc00::/7`).

Listing 51-1
```php
use Symfony\Component\HttpFoundation\Request;

// only trust proxy headers coming from this IP addresses
Request::setTrustedProxies(array('192.0.0.1', '10.0.0.0/8'));
```

Configuring Header Names

By default, the following proxy headers are trusted:

- `X-Forwarded-For` Used in *`getClientIp()`*[1];
- `X-Forwarded-Host` Used in *`getHost()`*[2];
- `X-Forwarded-Port` Used in *`getPort()`*[3];
- `X-Forwarded-Proto` Used in *`getScheme()`*[4] and *`isSecure()`*[5];

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getClientIp()
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getHost()
3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getPort()
4. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getScheme()

If your reverse proxy uses a different header name for any of these, you can configure that header name via *setTrustedHeaderName()*[6]:

Listing 51-2
```
Request::setTrustedHeaderName(Request::HEADER_CLIENT_IP, 'X-Proxy-For');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_HOST, 'X-Proxy-Host');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PORT, 'X-Proxy-Port');
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PROTO, 'X-Proxy-Proto');
```

Not Trusting certain Headers

By default, if you whitelist your proxy's IP address, then all four headers listed above are trusted. If you need to trust some of these headers but not others, you can do that as well:

Listing 51-3
```
// disables trusting the ``X-Forwarded-Proto`` header, the default header is used
Request::setTrustedHeaderName(Request::HEADER_CLIENT_PROTO, '');
```

5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#isSecure()
6. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#setTrustedHeaderName()

Chapter 52
The HttpKernel Component

> The HttpKernel component provides a structured process for converting a `Request` into a `Response` by making use of the EventDispatcher. It's flexible enough to create a full-stack framework (Symfony), a micro-framework (Silex) or an advanced CMS system (Drupal).

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/http-kernel` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/HttpKernel*[2]).

The Workflow of a Request

Every HTTP web interaction begins with a request and ends with a response. Your job as a developer is to create PHP code that reads the request information (e.g. the URL) and creates and returns a response (e.g. an HTML page or JSON string).

1. `https://packagist.org/packages/symfony/http-kernel`
2. `https://github.com/symfony/HttpKernel`

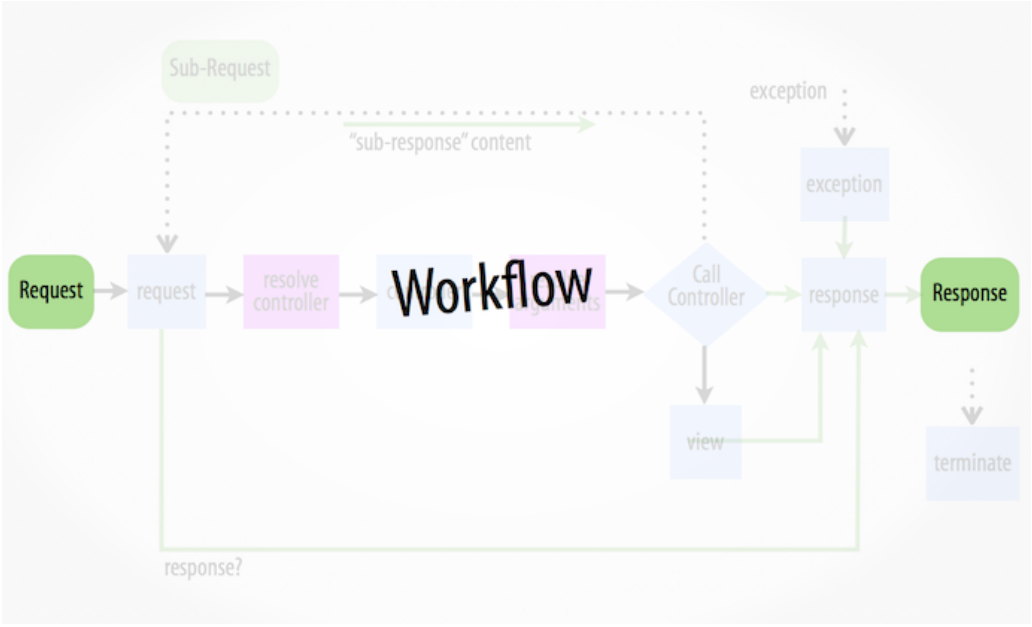
Typically, some sort of framework or system is built to handle all the repetitive tasks (e.g. routing, security, etc) so that a developer can easily build each *page* of the application. Exactly *how* these systems are built varies greatly. The HttpKernel component provides an interface that formalizes the process of starting with a request and creating the appropriate response. The component is meant to be the heart of any application or framework, no matter how varied the architecture of that system:

Listing 52-1

```php
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;

interface HttpKernelInterface
{
    // ...

    /**
     * @return Response A Response instance
     */
    public function handle(
        Request $request,
        $type = self::MASTER_REQUEST,
        $catch = true
    );
}
```

Internally, *`HttpKernel::handle()`*[3] - the concrete implementation of *`HttpKernelInterface::handle()`*[4] - defines a workflow that starts with a *`Request`*[5] and ends with a *`Response`*[6].

3. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernel.html#handle()
4. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernelInterface.html#handle()
5. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
6. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html



The exact details of this workflow are the key to understanding how the kernel (and the Symfony Framework or any other library that uses the kernel) works.

HttpKernel: Driven by Events

The `HttpKernel::handle()` method works internally by dispatching events. This makes the method both flexible, but also a bit abstract, since all the "work" of a framework/application built with HttpKernel is actually done in event listeners.

To help explain this process, this document looks at each step of the process and talks about how one specific implementation of the HttpKernel - the Symfony Framework - works.

Initially, using the *`HttpKernel`*[7] is really simple, and involves creating an *EventDispatcher* and a *controller resolver* (explained below). To complete your working kernel, you'll add more event listeners to the events discussed below:

Listing 52-2

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

// create the Request object
$request = Request::createFromGlobals();

$dispatcher = new EventDispatcher();
// ... add some event listeners

// create your controller resolver
$resolver = new ControllerResolver();
// instantiate the kernel
$kernel = new HttpKernel($dispatcher, $resolver);

// actually execute the kernel, which turns the request into a response
// by dispatching events, calling a controller, and returning the response
$response = $kernel->handle($request);
```

7. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernel.html

```
20
21 // send the headers and echo the content
22 $response->send();
23
24 // triggers the kernel.terminate event
25 $kernel->terminate($request, $response);
```

See "*A full Working Example*" for a more concrete implementation.

For general information on adding listeners to the events below, see *Creating an Event Listener*.

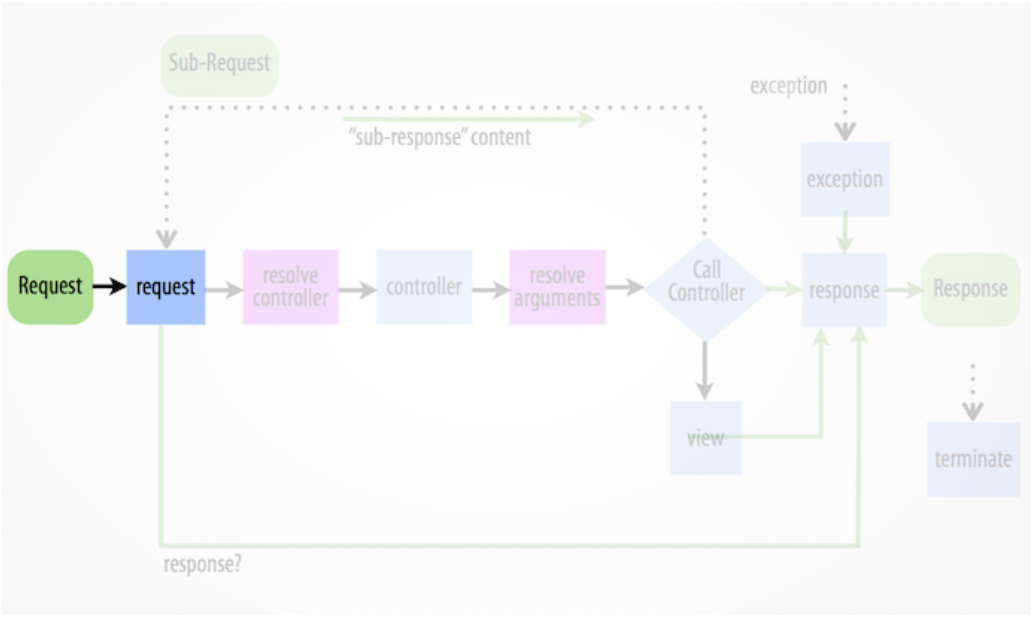
Fabien Potencier also wrote a wonderful series on using the HttpKernel component and other Symfony components to create your own framework. See *Create your own framework... on top of the Symfony2 Components*[8].

1) The `kernel.request` Event

Typical Purposes: To add more information to the `Request`, initialize parts of the system, or return a `Response` if possible (e.g. a security layer that denies access).

Kernel Events Information Table

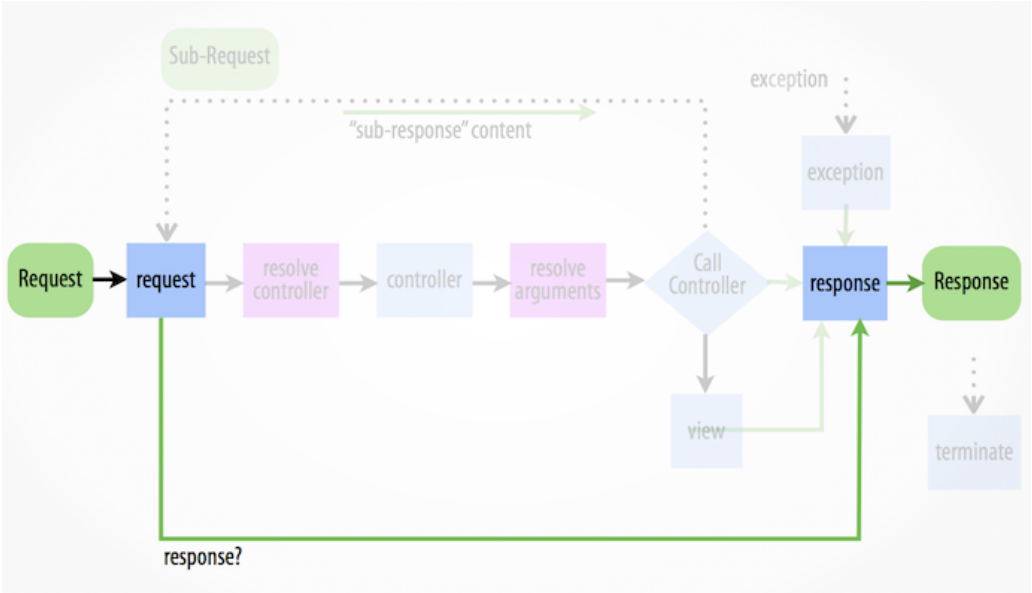
The first event that is dispatched inside *`HttpKernel::handle`*[9] is `kernel.request`, which may have a variety of different listeners.



Listeners of this event can be quite varied. Some listeners - such as a security listener - might have enough information to create a `Response` object immediately. For example, if a security listener determined that a user doesn't have access, that listener may return a *`RedirectResponse`*[10] to the login page or a 403 Access Denied response.

If a `Response` is returned at this stage, the process skips directly to the *kernel.response* event.

8. http://fabien.potencier.org/article/50/create-your-own-framework-on-top-of-the-symfony2-components-part-1
9. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernel.html#handle()
10. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/RedirectResponse.html



Other listeners simply initialize things or add more information to the request. For example, a listener might determine and set the locale on the `Request` object.

Another common listener is routing. A router listener may process the `Request` and determine the controller that should be rendered (see the next section). In fact, the `Request` object has an "*attributes*" bag which is a perfect spot to store this extra, application-specific data about the request. This means that if your router listener somehow determines the controller, it can store it on the `Request` attributes (which can be used by your controller resolver).

Overall, the purpose of the `kernel.request` event is either to create and return a `Response` directly, or to add information to the `Request` (e.g. setting the locale or setting some other information on the `Request` attributes).

When setting a response for the `kernel.request` event, the propagation is stopped. This means listeners with lower priority won't be executed.

`kernel.request` in the Symfony Framework

The most important listener to `kernel.request` in the Symfony Framework is the *`RouterListener`*[11]. This class executes the routing layer, which returns an *array* of information about the matched request, including the `_controller` and any placeholders that are in the route's pattern (e.g. `{slug}`). See *Routing component*.

This array of information is stored in the *`Request`*[12] object's `attributes` array. Adding the routing information here doesn't do anything yet, but is used next when resolving the controller.

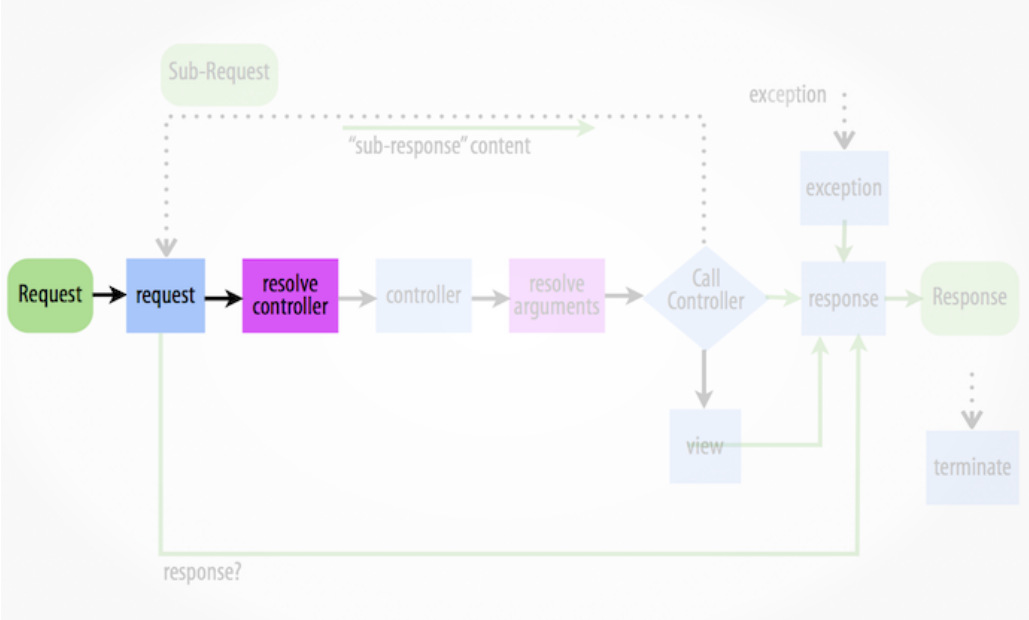
2) Resolve the Controller

Assuming that no `kernel.request` listener was able to create a `Response`, the next step in HttpKernel is to determine and prepare (i.e. resolve) the controller. The controller is the part of the end-application's

11. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/RouterListener.html
12. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html

code that is responsible for creating and returning the `Response` for a specific page. The only requirement is that it is a PHP callable - i.e. a function, method on an object, or a `Closure`.

But *how* you determine the exact controller for a request is entirely up to your application. This is the job of the "controller resolver" - a class that implements *`ControllerResolverInterface`*[13] and is one of the constructor arguments to `HttpKernel`.



Your job is to create a class that implements the interface and fill in its two methods: `getController` and `getArguments`. In fact, one default implementation already exists, which you can use directly or learn from: *`ControllerResolver`*[14]. This implementation is explained more in the sidebar below:

Listing 52-3

```php
namespace Symfony\Component\HttpKernel\Controller;

use Symfony\Component\HttpFoundation\Request;

interface ControllerResolverInterface
{
    public function getController(Request $request);

    public function getArguments(Request $request, $controller);
}
```

Internally, the `HttpKernel::handle` method first calls *`getController()`*[15] on the controller resolver. This method is passed the `Request` and is responsible for somehow determining and returning a PHP callable (the controller) based on the request's information.

The second method, *`getArguments()`*[16], will be called after another event - `kernel.controller` - is dispatched.

13. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html
14. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
15. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getController()
16. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()

Resolving the Controller in the Symfony Framework

The Symfony Framework uses the built-in *`ControllerResolver`*[17] class (actually, it uses a sub-class with some extra functionality mentioned below). This class leverages the information that was placed on the `Request` object's `attributes` property during the `RouterListener`.

getController

The `ControllerResolver` looks for a `_controller` key on the `Request` object's attributes property (recall that this information is typically placed on the `Request` via the `RouterListener`). This string is then transformed into a PHP callable by doing the following:

1. The `AcmeDemoBundle:Default:index` format of the `_controller` key is changed to another string that contains the full class and method name of the controller by following the convention used in Symfony - e.g. `Acme\DemoBundle\Controller\DefaultController::indexAction`. This transformation is specific to the *`ControllerResolver`*[18] sub-class used by the Symfony Framework.
2. A new instance of your controller class is instantiated with no constructor arguments.
3. If the controller implements *`ContainerAwareInterface`*[19], `setContainer` is called on the controller object and the container is passed to it. This step is also specific to the *`ControllerResolver`*[20] sub-class used by the Symfony Framework.

   There are also a few other variations on the above process (e.g. if you're registering your controllers as services).

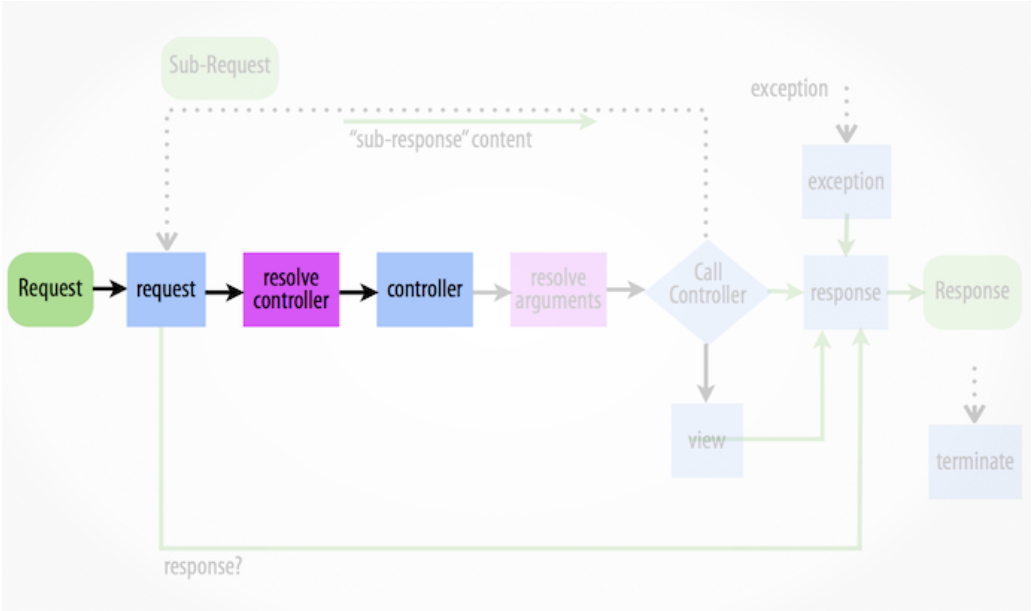
3) The `kernel.controller` Event

Typical Purposes: Initialize things or change the controller just before the controller is executed.

Kernel Events Information Table

After the controller callable has been determined, `HttpKernel::handle` dispatches the `kernel.controller` event. Listeners to this event might initialize some part of the system that needs to be initialized after certain things have been determined (e.g. the controller, routing information) but before the controller is executed. For some examples, see the Symfony section below.

17. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
18. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html
19. http://api.symfony.com/2.3/Symfony/Component/DependencyInjection/ContainerAwareInterface.html
20. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html



Listeners to this event can also change the controller callable completely by calling *`FilterControllerEvent::setController`*[21] on the event object that's passed to listeners on this event.

`kernel.controller` in the Symfony Framework

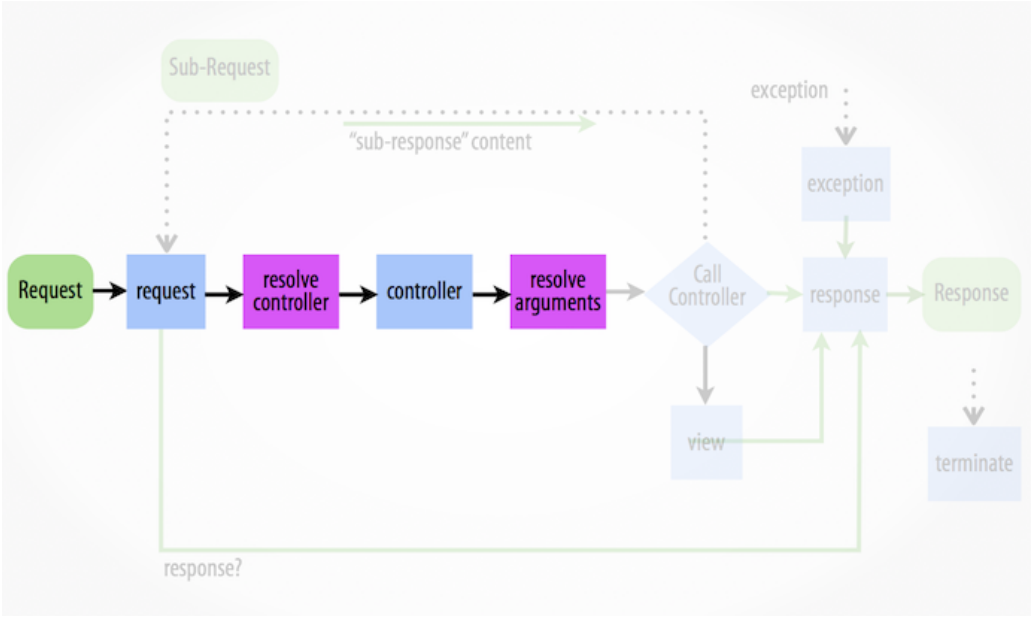
There are a few minor listeners to the `kernel.controller` event in the Symfony Framework, and many deal with collecting profiler data when the profiler is enabled.

One interesting listener comes from the *SensioFrameworkExtraBundle*[22], which is packaged with the Symfony Standard Edition. This listener's *@ParamConverter*[23] functionality allows you to pass a full object (e.g. a `Post` object) to your controller instead of a scalar value (e.g. an `id` parameter that was on your route). The listener - `ParamConverterListener` - uses reflection to look at each of the arguments of the controller and tries to use different methods to convert those to objects, which are then stored in the `attributes` property of the `Request` object. Read the next section to see why this is important.

4) Getting the Controller Arguments

Next, `HttpKernel::handle` calls *`getArguments()`*[24]. Remember that the controller returned in `getController` is a callable. The purpose of `getArguments` is to return the array of arguments that should be passed to that controller. Exactly how this is done is completely up to your design, though the built-in *`ControllerResolver`*[25] is a good example.

21. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html#setController()
22. http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html
23. http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html
24. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()
25. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolver.html



At this point the kernel has a PHP callable (the controller) and an array of arguments that should be passed when executing that callable.

Getting the Controller Arguments in the Symfony Framework

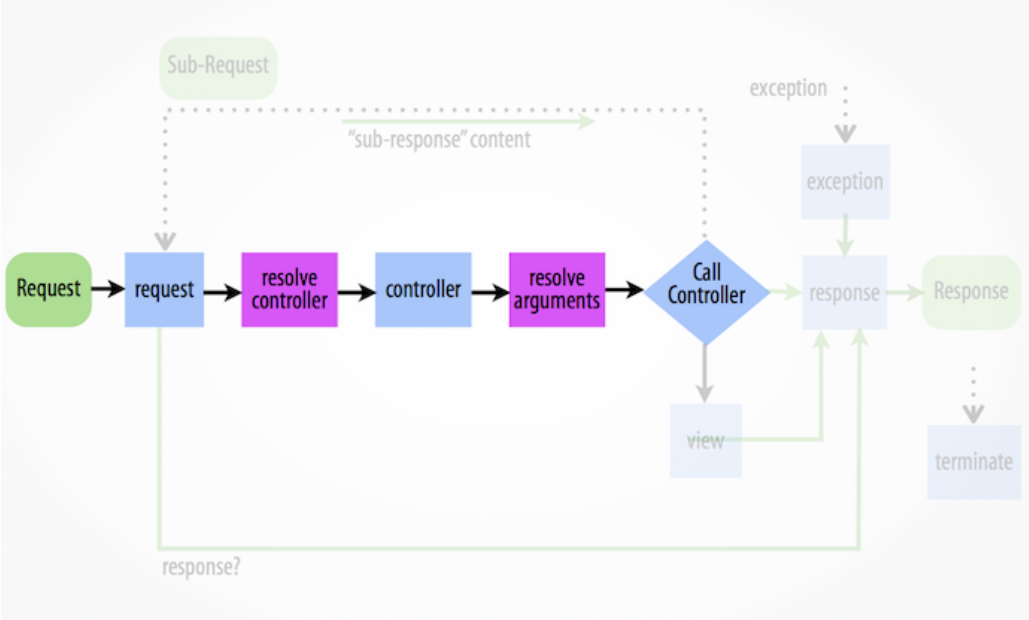
Now that you know exactly what the controller callable (usually a method inside a controller object) is, the `ControllerResolver` uses *reflection*[26] on the callable to return an array of the *names* of each of the arguments. It then iterates over each of these arguments and uses the following tricks to determine which value should be passed for each argument:

1. If the `Request` attributes bag contains a key that matches the name of the argument, that value is used. For example, if the first argument to a controller is `$slug`, and there is a `slug` key in the `Request attributes` bag, that value is used (and typically this value came from the `RouterListener`).
2. If the argument in the controller is type-hinted with Symfony's *`Request`*[27] object, then the `Request` is passed in as the value.

5) Calling the Controller

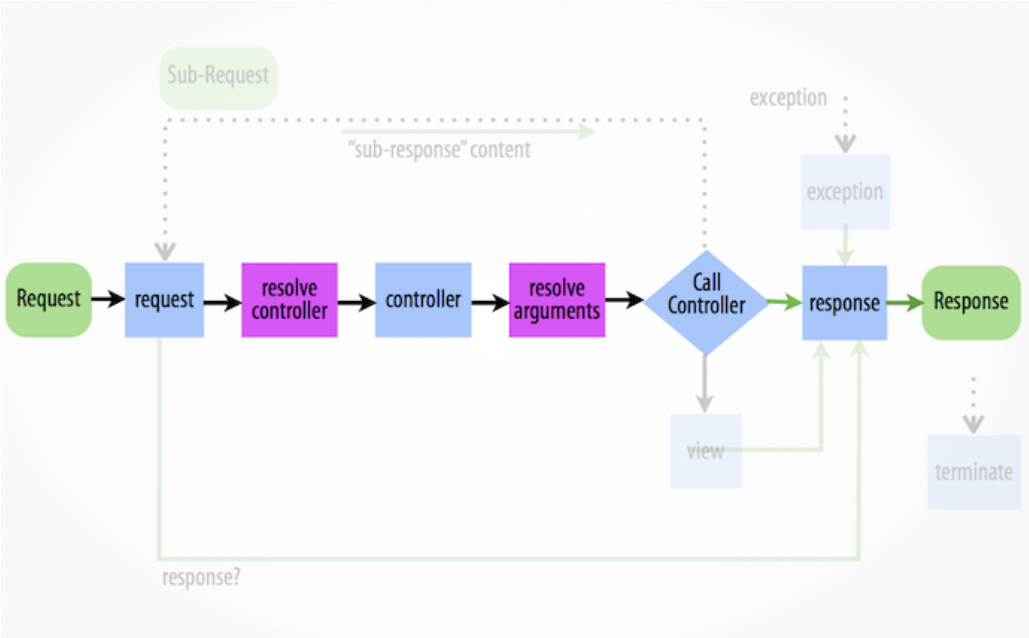
The next step is simple! `HttpKernel::handle` executes the controller.

26. http://php.net/manual/en/book.reflection.php
27. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html



The job of the controller is to build the response for the given resource. This could be an HTML page, a JSON string or anything else. Unlike every other part of the process so far, this step is implemented by the "end-developer", for each page that is built.

Usually, the controller will return a `Response` object. If this is true, then the work of the kernel is just about done! In this case, the next step is the *kernel.response* event.



But if the controller returns anything besides a `Response`, then the kernel has a little bit more work to do - *kernel.view* (since the end goal is *always* to generate a `Response` object).

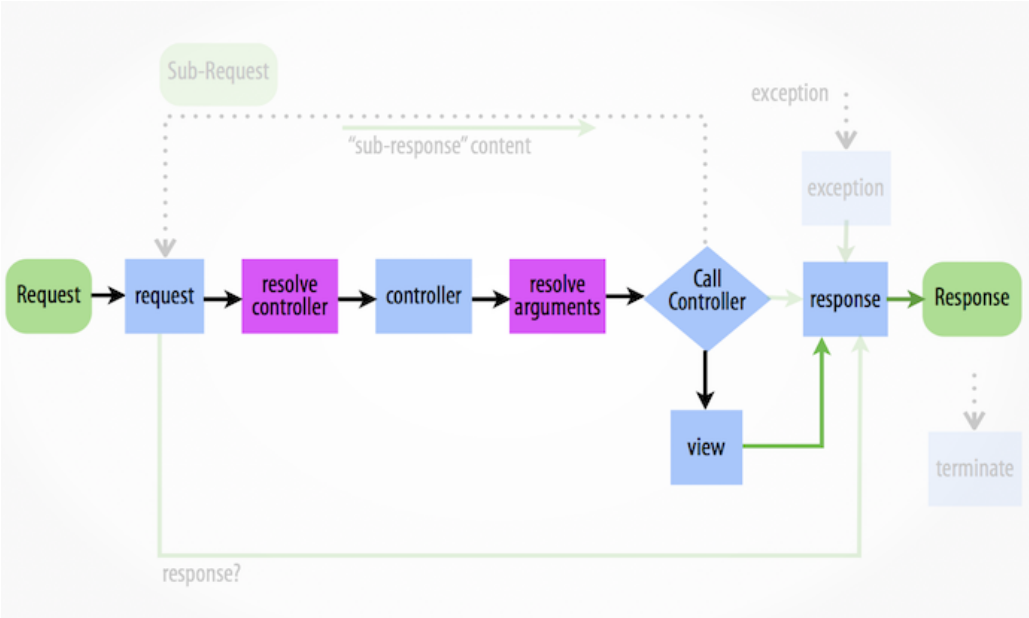
A controller must return *something*. If a controller returns `null`, an exception will be thrown immediately.

6) The `kernel.view` Event

Typical Purposes: Transform a non-`Response` return value from a controller into a `Response`

Kernel Events Information Table

If the controller doesn't return a `Response` object, then the kernel dispatches another event - `kernel.view`. The job of a listener to this event is to use the return value of the controller (e.g. an array of data or an object) to create a `Response`.



This can be useful if you want to use a "view" layer: instead of returning a `Response` from the controller, you return data that represents the page. A listener to this event could then use this data to create a `Response` that is in the correct format (e.g HTML, JSON, etc).

At this stage, if no listener sets a response on the event, then an exception is thrown: either the controller *or* one of the view listeners must always return a `Response`.

When setting a response for the `kernel.view` event, the propagation is stopped. This means listeners with lower priority won't be executed.

`kernel.view` in the Symfony Framework

There is no default listener inside the Symfony Framework for the `kernel.view` event. However, one core bundle - *SensioFrameworkExtraBundle*[28] - *does* add a listener to this event. If your controller returns an array, and you place the *@Template*[29] annotation above the controller, then this listener renders a template, passes the array you returned from your controller to that template, and creates a `Response` containing the returned content from that template.

Additionally, a popular community bundle *FOSRestBundle*[30] implements a listener on this event which aims to give you a robust view layer capable of using a single controller to return many different content-type responses (e.g. HTML, JSON, XML, etc).

28. `http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html`
29. `http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/view.html`
30. `https://github.com/friendsofsymfony/FOSRestBundle`

7) The `kernel.response` Event

Typical Purposes: Modify the `Response` object just before it is sent

Kernel Events Information Table

The end goal of the kernel is to transform a `Request` into a `Response`. The `Response` might be created during the *kernel.request* event, returned from the *controller*, or returned by one of the listeners to the *kernel.view* event.

Regardless of who creates the `Response`, another event - `kernel.response` is dispatched directly afterwards. A typical listener to this event will modify the `Response` object in some way, such as modifying headers, adding cookies, or even changing the content of the `Response` itself (e.g. injecting some JavaScript before the end `</body>` tag of an HTML response).

After this event is dispatched, the final `Response` object is returned from *`handle()`*[31]. In the most typical use-case, you can then call the *`send()`*[32] method, which sends the headers and prints the `Response` content.

`kernel.response` in the Symfony Framework

There are several minor listeners on this event inside the Symfony Framework, and most modify the response in some way. For example, the *`WebDebugToolbarListener`*[33] injects some JavaScript at the bottom of your page in the `dev` environment which causes the web debug toolbar to be displayed. Another listener, *`ContextListener`*[34] serializes the current user's information into the session so that it can be reloaded on the next request.

8) The `kernel.terminate` Event

Typical Purposes: To perform some "heavy" action after the response has been streamed to the user

Kernel Events Information Table

The final event of the HttpKernel process is `kernel.terminate` and is unique because it occurs *after* the `HttpKernel::handle` method, and after the response is sent to the user. Recall from above, then the code that uses the kernel, ends like this:

Listing 52-4
```
// send the headers and echo the content
$response->send();

// triggers the kernel.terminate event
$kernel->terminate($request, $response);
```

As you can see, by calling `$kernel->terminate` after sending the response, you will trigger the `kernel.terminate` event where you can perform certain actions that you may have delayed in order to return the response as quickly as possible to the client (e.g. sending emails).

Internally, the HttpKernel makes use of the *`fastcgi_finish_request`*[35] PHP function. This means that at the moment, only the *PHP FPM*[36] server API is able to send a response to the client while the server's PHP process still performs some tasks. With all other server APIs, listeners to `kernel.terminate` are still executed, but the response is not sent to the client until they are all completed.

31. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernel.html#handle()
32. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html#send()
33. http://api.symfony.com/2.3/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
34. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall/ContextListener.html

Using the `kernel.terminate` event is optional, and should only be called if your kernel implements *`TerminableInterface`*[37].

`kernel.terminate` in the Symfony Framework

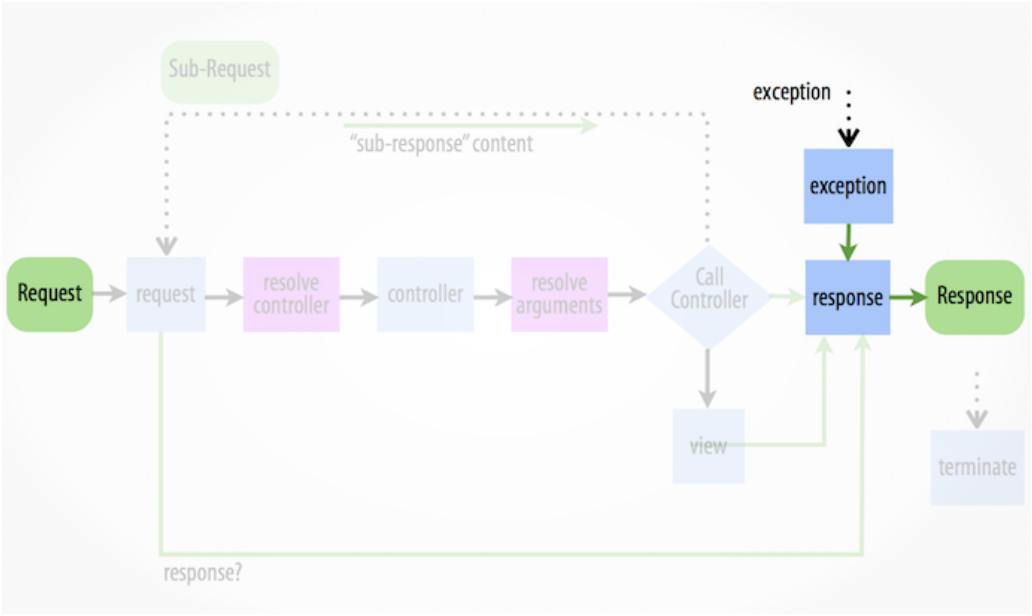
If you use the SwiftmailerBundle with Symfony and use `memory` spooling, then the *EmailSenderListener*[38] is activated, which actually delivers any emails that you scheduled to send during the request.

Handling Exceptions: the `kernel.exception` Event

Typical Purposes: Handle some type of exception and create an appropriate `Response` to return for the exception

Kernel Events Information Table

If an exception is thrown at any point inside `HttpKernel::handle`, another event - `kernel.exception` is thrown. Internally, the body of the `handle` function is wrapped in a try-catch block. When any exception is thrown, the `kernel.exception` event is dispatched so that your system can somehow respond to the exception.



Each listener to this event is passed a *`GetResponseForExceptionEvent`*[39] object, which you can use to access the original exception via the *`getException()`*[40] method. A typical listener on this event will check for a certain type of exception and create an appropriate error `Response`.

For example, to generate a 404 page, you might throw a special type of exception and then add a listener on this event that looks for this exception and creates and returns a 404 `Response`. In fact, the HttpKernel

35. http://php.net/manual/en/function.fastcgi-finish-request.php
36. http://php.net/manual/en/install.fpm.php
37. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/TerminableInterface.html
38. https://github.com/symfony/SwiftmailerBundle/blob/master/EventListener/EmailSenderListener.php
39. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
40. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html#getException()

component comes with an *ExceptionListener*[41], which if you choose to use, will do this and more by default (see the sidebar below for more details).

When setting a response for the `kernel.exception` event, the propagation is stopped. This means listeners with lower priority won't be executed.

`kernel.exception` in the Symfony Framework

There are two main listeners to `kernel.exception` when using the Symfony Framework.

ExceptionListener in HttpKernel

The first comes core to the HttpKernel component and is called *ExceptionListener*[42]. The listener has several goals:

1. The thrown exception is converted into a *FlattenException*[43] object, which contains all the information about the request, but which can be printed and serialized.
2. If the original exception implements *HttpExceptionInterface*[44], then `getStatusCode` and `getHeaders` are called on the exception and used to populate the headers and status code of the `FlattenException` object. The idea is that these are used in the next step when creating the final response.
3. A controller is executed and passed the flattened exception. The exact controller to render is passed as a constructor argument to this listener. This controller will return the final `Response` for this error page.

ExceptionListener in Security

The other important listener is the *ExceptionListener*[45]. The goal of this listener is to handle security exceptions and, when appropriate, *help* the user to authenticate (e.g. redirect to the login page).

Creating an Event Listener

As you've seen, you can create and attach event listeners to any of the events dispatched during the `HttpKernel::handle` cycle. Typically a listener is a PHP class with a method that's executed, but it can be anything. For more information on creating and attaching event listeners, see *The EventDispatcher Component*.

The name of each of the "kernel" events is defined as a constant on the *KernelEvents*[46] class. Additionally, each event listener is passed a single argument, which is some sub-class of *KernelEvent*[47]. This object contains information about the current state of the system and each event has their own event object:

Name	KernelEvents Constant	Argument Passed to the Listener
kernel.request	`KernelEvents::REQUEST`	*GetResponseEvent*[48]
kernel.controller	`KernelEvents::CONTROLLER`	*FilterControllerEvent*[49]

41. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
42. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
43. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Exception/FlattenException.html
44. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.html
45. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall/ExceptionListener.html
46. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelEvents.html
47. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/KernelEvent.html
48. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseEvent.html

Name	KernelEvents Constant	Argument Passed to the Listener
kernel.view	KernelEvents::VIEW	*GetResponseForControllerResultEvent*[50]
kernel.response	KernelEvents::RESPONSE	*FilterResponseEvent*[51]
kernel.terminate	KernelEvents::TERMINATE	*PostResponseEvent*[52]
kernel.exception	KernelEvents::EXCEPTION	*GetResponseForExceptionEvent*[53]

A full Working Example

When using the HttpKernel component, you're free to attach any listeners to the core events and use any controller resolver that implements the *ControllerResolverInterface*[54]. However, the HttpKernel component comes with some built-in listeners and a built-in ControllerResolver that can be used to create a working example:

Listing 52-5

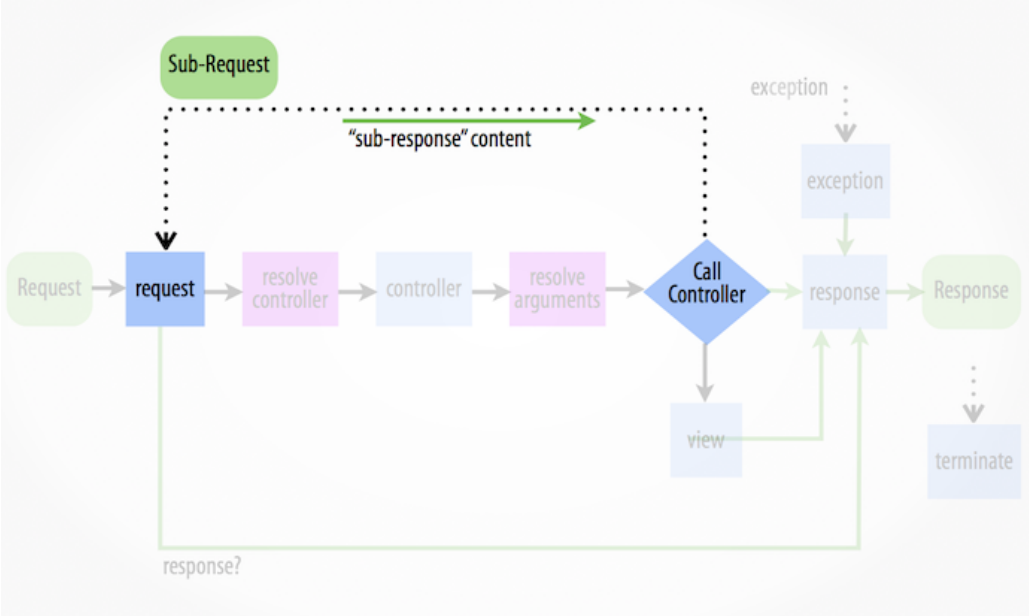
```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;
use Symfony\Component\HttpKernel\EventListener\RouterListener;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;

$routes = new RouteCollection();
$routes->add('hello', new Route('/hello/{name}', array(
        '_controller' => function (Request $request) {
            return new Response(
                sprintf("Hello %s", $request->get('name'))
            );
        }
    )
));

$request = Request::createFromGlobals();

$matcher = new UrlMatcher($routes, new RequestContext());

$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new RouterListener($matcher));

$resolver = new ControllerResolver();
$kernel = new HttpKernel($dispatcher, $resolver);

$response = $kernel->handle($request);
$response->send();
```

49. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html
50. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
51. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html
52. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/PostResponseEvent.html
53. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
54. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html

```
34
35 $kernel->terminate($request, $response);
```

Sub Requests

In addition to the "main" request that's sent into `HttpKernel::handle`, you can also send so-called "sub request". A sub request looks and acts like any other request, but typically serves to render just one small portion of a page instead of a full page. You'll most commonly make sub-requests from your controller (or perhaps from inside a template, that's being rendered by your controller).



To execute a sub request, use `HttpKernel::handle`, but change the second argument as follows:

Listing 52-6
```
1  use Symfony\Component\HttpFoundation\Request;
2  use Symfony\Component\HttpKernel\HttpKernelInterface;
3
4  // ...
5
6  // create some other request manually as needed
7  $request = new Request();
8  // for example, possibly set its _controller manually
9  $request->attributes->set('_controller', '...');
10
11 $response = $kernel->handle($request, HttpKernelInterface::SUB_REQUEST);
12 // do something with this response
```

This creates another full request-response cycle where this new `Request` is transformed into a `Response`. The only difference internally is that some listeners (e.g. security) may only act upon the master request. Each listener is passed some sub-class of *`KernelEvent`*[55], whose *`getRequestType()`*[56] can be used to figure out if the current request is a "master" or "sub" request.

For example, a listener that only needs to act on the master request may look like this:

55. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/KernelEvent.html
56. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/KernelEvent.html#getRequestType()

Listing 52-7

```php
use Symfony\Component\HttpKernel\HttpKernelInterface;
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
        return;
    }

    // ...
}
```

Chapter 53
The Intl Component

> A PHP replacement layer for the C *intl extension*[1] that also provides access to the localization data of the *ICU library*[2].

New in version 2.3: The Intl component was introduced in Symfony 2.3. In earlier versions of Symfony, you should use the Locale component instead.

The replacement layer is limited to the locale "en". If you want to use other locales, you should *install the intl extension*[3] instead.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/intl` on *Packagist*[4]);
- Using the official Git repository (*https://github.com/symfony/Intl*[5]).

If you install the component via Composer, the following classes and functions of the intl extension will be automatically provided if the intl extension is not loaded:

- *`Collator`*[6]
- *`IntlDateFormatter`*[7]
- *`Locale`*[8]

1. http://www.php.net/manual/en/book.intl.php
2. http://site.icu-project.org/
3. http://www.php.net/manual/en/intl.setup.php
4. https://packagist.org/packages/symfony/intl
5. https://github.com/symfony/Intl
6. http://php.net/manual/en/class.collator.php
7. http://php.net/manual/en/class.intldateformatter.php

- *NumberFormatter*[9]
- *intl_error_name*[10]
- *intl_is_failure*[11]
- *intl_get_error_code*[12]
- *intl_get_error_message*[13]

When the intl extension is not available, the following classes are used to replace the intl classes:

- *Collator*[14]
- *IntlDateFormatter*[15]
- *Locale*[16]
- *NumberFormatter*[17]
- *IntlGlobals*[18]

Composer automatically exposes these classes in the global namespace.

If you don't use Composer but the *Symfony ClassLoader component*, you need to expose them manually by adding the following lines to your autoload code:

Listing 53-1
```php
if (!function_exists('intl_is_failure')) {
    require '/path/to/Icu/Resources/stubs/functions.php';

    $loader->registerPrefixFallback('/path/to/Icu/Resources/stubs');
}
```

8. http://php.net/manual/en/class.locale.php
9. http://php.net/manual/en/class.numberformatter.php
10. http://php.net/manual/en/function.intl-error-name.php
11. http://php.net/manual/en/function.intl-is-failure.php
12. http://php.net/manual/en/function.intl-get-error-code.php
13. http://php.net/manual/en/function.intl-get-error-message.php
14. http://api.symfony.com/2.3/Symfony/Component/Intl/Collator/Collator.html
15. http://api.symfony.com/2.3/Symfony/Component/Intl/DateFormatter/IntlDateFormatter.html
16. http://api.symfony.com/2.3/Symfony/Component/Intl/Locale/Locale.html
17. http://api.symfony.com/2.3/Symfony/Component/Intl/NumberFormatter/NumberFormatter.html
18. http://api.symfony.com/2.3/Symfony/Component/Intl/Globals/IntlGlobals.html

ICU and Deployment Problems

These deployment problems only affect the following Symfony versions: 2.3.0 to 2.3.20 versions, any 2.4.x version and 2.5.0 to 2.5.5 versions.

The intl extension internally uses the *ICU library*[19] to obtain localization data such as number formats in different languages, country names and more. To make this data accessible to userland PHP libraries, Symfony ships a copy in the *Icu component*[20].

Depending on the ICU version compiled with your intl extension, a matching version of that component needs to be installed. It sounds complicated, but usually Composer does this for you automatically:

- 1.0.*: when the intl extension is not available
- 1.1.*: when intl is compiled with ICU 4.0 or higher
- 1.2.*: when intl is compiled with ICU 4.4 or higher

These versions are important when you deploy your application to a **server with a lower ICU version** than your development machines, because deployment will fail if:

- the development machines are compiled with ICU 4.4 or higher, but the server is compiled with a lower ICU version than 4.4;
- the intl extension is available on the development machines but not on the server.

For example, consider that your development machines ship ICU 4.8 and the server ICU 4.2. When you run `composer update` on the development machine, version 1.2.* of the Icu component will be installed. But after deploying the application, `composer install` will fail with the following error:

Listing 53-2
```
$ composer install
Loading composer repositories with package information
Installing dependencies from lock file
Your requirements could not be resolved to an installable set of packages.

  Problem 1
    - symfony/icu 1.2.x requires lib-icu >=4.4 -> the requested linked
      library icu has the wrong version installed or is missing from your
      system, make sure to have the extension providing it.
```

The error tells you that the requested version of the Icu component, version 1.2, is not compatible with PHP's ICU version 4.2.

One solution to this problem is to run `composer update` instead of `composer install`. It is highly recommended **not** to do this. The `update` command will install the latest versions of each Composer dependency to your production server and potentially break the application.

A better solution is to fix your composer.json to the version required by the production server. First, determine the ICU version on the server:

Listing 53-3
```
$ php -i | grep ICU
ICU version => 4.2.1
```

Then fix the Icu component in your `composer.json` file to a matching version:

Listing 53-4

19. http://site.icu-project.org/
20. https://packagist.org/packages/symfony/icu

```
"require: {
    "symfony/icu": "1.1.*"
}
```

Set the version to

- "1.0.*" if the server does not have the intl extension installed;
- "1.1.*" if the server is compiled with ICU 4.2 or lower.

Finally, run `composer update symfony/icu` on your development machine, test extensively and deploy again. The installation of the dependencies will now succeed.

Writing and Reading Resource Bundles

The *ResourceBundle*[21] class is not currently supported by this component. Instead, it includes a set of readers and writers for reading and writing arrays (or array-like objects) from/to resource bundle files. The following classes are supported:

- TextBundleWriter
- PhpBundleWriter
- BinaryBundleReader
- PhpBundleReader
- BufferedBundleReader
- StructuredBundleReader

Continue reading if you are interested in how to use these classes. Otherwise skip this section and jump to Accessing ICU Data.

TextBundleWriter

The *TextBundleWriter*[22] writes an array or an array-like object to a plain-text resource bundle. The resulting .txt file can be converted to a binary .res file with the *BundleCompiler*[23] class:

Listing 53-5
```php
use Symfony\Component\Intl\ResourceBundle\Writer\TextBundleWriter;
use Symfony\Component\Intl\ResourceBundle\Compiler\BundleCompiler;

$writer = new TextBundleWriter();
$writer->write('/path/to/bundle', 'en', array(
    'Data' => array(
        'entry1',
        'entry2',
        // ...
    ),
));

$compiler = new BundleCompiler();
$compiler->compile('/path/to/bundle', '/path/to/binary/bundle');
```

21. http://php.net/manual/en/class.resourcebundle.php
22. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Writer/TextBundleWriter.html
23. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html

The command "genrb" must be available for the *BundleCompiler*[24] to work. If the command is located in a non-standard location, you can pass its path to the *BundleCompiler*[25] constructor.

PhpBundleWriter

The *PhpBundleWriter*[26] writes an array or an array-like object to a .php resource bundle:

Listing 53-6
```php
use Symfony\Component\Intl\ResourceBundle\Writer\PhpBundleWriter;

$writer = new PhpBundleWriter();
$writer->write('/path/to/bundle', 'en', array(
    'Data' => array(
        'entry1',
        'entry2',
        // ...
    ),
));
```

BinaryBundleReader

The *BinaryBundleReader*[27] reads binary resource bundle files and returns an array or an array-like object. This class currently only works with the *intl extension*[28] installed:

Listing 53-7
```php
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;

$reader = new BinaryBundleReader();
$data = $reader->read('/path/to/bundle', 'en');

echo $data['Data']['entry1'];
```

PhpBundleReader

The *PhpBundleReader*[29] reads resource bundles from .php files and returns an array or an array-like object:

Listing 53-8
```php
use Symfony\Component\Intl\ResourceBundle\Reader\PhpBundleReader;

$reader = new PhpBundleReader();
$data = $reader->read('/path/to/bundle', 'en');

echo $data['Data']['entry1'];
```

BufferedBundleReader

The *BufferedBundleReader*[30] wraps another reader, but keeps the last N reads in a buffer, where N is a buffer size passed to the constructor:

---

24. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html
25. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Compiler/BundleCompiler.html
26. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Writer/PhpBundleWriter.html
27. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/BinaryBundleReader.html
28. http://www.php.net/manual/en/book.intl.php
29. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/PhpBundleReader.html
30. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/BufferedBundleReader.html

Listing 53-9
```php
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;
use Symfony\Component\Intl\ResourceBundle\Reader\BufferedBundleReader;

$reader = new BufferedBundleReader(new BinaryBundleReader(), 10);

// actually reads the file
$data = $reader->read('/path/to/bundle', 'en');

// returns data from the buffer
$data = $reader->read('/path/to/bundle', 'en');

// actually reads the file
$data = $reader->read('/path/to/bundle', 'fr');
```

StructuredBundleReader

The *StructuredBundleReader*[31] wraps another reader and offers a *readEntry()*[32] method for reading an entry of the resource bundle without having to worry whether array keys are set or not. If a path cannot be resolved, `null` is returned:

Listing 53-10
```php
use Symfony\Component\Intl\ResourceBundle\Reader\BinaryBundleReader;
use Symfony\Component\Intl\ResourceBundle\Reader\StructuredBundleReader;

$reader = new StructuredBundleReader(new BinaryBundleReader());

$data = $reader->read('/path/to/bundle', 'en');

// Produces an error if the key "Data" does not exist
echo $data['Data']['entry1'];

// Returns null if the key "Data" does not exist
echo $reader->readEntry('/path/to/bundle', 'en', array('Data', 'entry1'));
```

Additionally, the *readEntry()*[33] method resolves fallback locales. For example, the fallback locale of "en_GB" is "en". For single-valued entries (strings, numbers etc.), the entry will be read from the fallback locale if it cannot be found in the more specific locale. For multi-valued entries (arrays), the values of the more specific and the fallback locale will be merged. In order to suppress this behavior, the last parameter `$fallback` can be set to `false`:

Listing 53-11
```php
echo $reader->readEntry(
    '/path/to/bundle',
    'en',
    array('Data', 'entry1'),
    false
);
```

Accessing ICU Data

The ICU data is located in several "resource bundles". You can access a PHP wrapper of these bundles through the static *Intl*[34] class. At the moment, the following data is supported:

31. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReader.html
32. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReaderInterface.html#readEntry()
33. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/Reader/StructuredBundleReaderInterface.html#readEntry()

- Language and Script Names
- Country Names
- Locales
- Currencies

Language and Script Names

The translations of language and script names can be found in the language bundle:

Listing 53-12
```php
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$languages = Intl::getLanguageBundle()->getLanguageNames();
// => array('ab' => 'Abkhazian', ...)

$language = Intl::getLanguageBundle()->getLanguageName('de');
// => 'German'

$language = Intl::getLanguageBundle()->getLanguageName('de', 'AT');
// => 'Austrian German'

$scripts = Intl::getLanguageBundle()->getScriptNames();
// => array('Arab' => 'Arabic', ...)

$script = Intl::getLanguageBundle()->getScriptName('Hans');
// => 'Simplified'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 53-13
```php
$languages = Intl::getLanguageBundle()->getLanguageNames('de');
// => array('ab' => 'Abchasisch', ...)
```

Country Names

The translations of country names can be found in the region bundle:

Listing 53-14
```php
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$countries = Intl::getRegionBundle()->getCountryNames();
// => array('AF' => 'Afghanistan', ...)

$country = Intl::getRegionBundle()->getCountryName('GB');
// => 'United Kingdom'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 53-15
```php
$countries = Intl::getRegionBundle()->getCountryNames('de');
// => array('AF' => 'Afghanistan', ...)
```

34. http://api.symfony.com/2.3/Symfony/Component/Intl/Intl.html

Locales

The translations of locale names can be found in the locale bundle:

Listing 53-16
```php
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$locales = Intl::getLocaleBundle()->getLocaleNames();
// => array('af' => 'Afrikaans', ...)

$locale = Intl::getLocaleBundle()->getLocaleName('zh_Hans_MO');
// => 'Chinese (Simplified, Macau SAR China)'
```

All methods accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 53-17
```php
$locales = Intl::getLocaleBundle()->getLocaleNames('de');
// => array('af' => 'Afrikaans', ...)
```

Currencies

The translations of currency names and other currency-related information can be found in the currency bundle:

Listing 53-18
```php
use Symfony\Component\Intl\Intl;

\Locale::setDefault('en');

$currencies = Intl::getCurrencyBundle()->getCurrencyNames();
// => array('AFN' => 'Afghan Afghani', ...)

$currency = Intl::getCurrencyBundle()->getCurrencyName('INR');
// => 'Indian Rupee'

$symbol = Intl::getCurrencyBundle()->getCurrencySymbol('INR');
// => '?'

$fractionDigits = Intl::getCurrencyBundle()->getFractionDigits('INR');
// => 2

$roundingIncrement = Intl::getCurrencyBundle()->getRoundingIncrement('INR');
// => 0
```

All methods (except for *getFractionDigits()*[35] and *getRoundingIncrement()*[36]) accept the translation locale as the last, optional parameter, which defaults to the current default locale:

Listing 53-19
```php
$currencies = Intl::getCurrencyBundle()->getCurrencyNames('de');
// => array('AFN' => 'Afghanische Afghani', ...)
```

That's all you need to know for now. Have fun coding!

35. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/CurrencyBundleInterface.html#getFractionDigits()
36. http://api.symfony.com/2.3/Symfony/Component/Intl/ResourceBundle/CurrencyBundleInterface.html#getRoundingIncrement()

Chapter 54
The OptionsResolver Component

> The OptionsResolver component helps you configure objects with option arrays. It supports default values, option constraints and lazy options.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/options-resolver` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/OptionsResolver*[2]).

Usage

Imagine you have a `Mailer` class which has 2 options: `host` and `password`. These options are going to be handled by the OptionsResolver Component.

First, create the `Mailer` class:

Listing 54-1
```php
class Mailer
{
    protected $options;

    public function __construct(array $options = array())
    {
    }
}
```

1. https://packagist.org/packages/symfony/options-resolver
2. https://github.com/symfony/OptionsResolver

You could of course set the `$options` value directly on the property. Instead, use the *`OptionsResolver`*[3] class and let it resolve the options by calling *`resolve()`*[4]. The advantages of doing this will become more obvious as you continue:

Listing 54-2
```php
use Symfony\Component\OptionsResolver\OptionsResolver;

// ...
public function __construct(array $options = array())
{
    $resolver = new OptionsResolver();

    $this->options = $resolver->resolve($options);
}
```

The options property now is a well defined array with all resolved options readily available:

Listing 54-3
```php
// ...
public function sendMail($from, $to)
{
    $mail = ...;
    $mail->setHost($this->options['host']);
    $mail->setUsername($this->options['username']);
    $mail->setPassword($this->options['password']);
    // ...
}
```

Configuring the OptionsResolver

Now, try to actually use the class:

Listing 54-4
```php
$mailer = new Mailer(array(
    'host'     => 'smtp.example.org',
    'username' => 'user',
    'password' => 'pa$$word',
));
```

Right now, you'll receive a *`InvalidOptionsException`*[5], which tells you that the options `host` and `password` do not exist. This is because you need to configure the `OptionsResolver` first, so it knows which options should be resolved.

To check if an option exists, you can use the *`isKnown()`*[6] function.

A best practice is to put the configuration in a method (e.g. `configureOptions`). You call this method in the constructor to configure the `OptionsResolver` class:

Listing 54-5

3. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html
4. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#resolve()
5. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/Exception/InvalidOptionsException.html
6. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#isKnown()

```php
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class Mailer
{
    protected $options;

    public function __construct(array $options = array())
    {
        $resolver = new OptionsResolver();
        $this->configureOptions($resolver);

        $this->options = $resolver->resolve($options);
    }

    protected function configureOptions(OptionsResolverInterface $resolver)
    {
        // ... configure the resolver, you will learn this
        // in the sections below
    }
}
```

Set default Values

Most of the options have a default value. You can configure these options by calling *`setDefaults()`*[7]:

Listing 54-6
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setDefaults(array(
        'username' => 'root',
    ));
}
```

This would add an option - `username` - and give it a default value of `root`. If the user passes in a `username` option, that value will override this default. You don't need to configure `username` as an optional option.

Required Options

The `host` option is required: the class can't work without it. You can set the required options by calling *`setRequired()`*[8]:

Listing 54-7
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setRequired(array('host'));
}
```

You are now able to use the class without errors:

Listing 54-8

7. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setDefaults()
8. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setRequired()

```php
$mailer = new Mailer(array(
    'host' => 'smtp.example.org',
));

echo $mailer->getHost(); // 'smtp.example.org'
```

If you don't pass a required option, a *MissingOptionsException*[9] will be thrown.

To determine if an option is required, you can use the *isRequired()*[10] method.

Optional Options

Sometimes, an option can be optional (e.g. the `password` option in the `Mailer` class), but it doesn't have a default value. You can configure these options by calling *setOptional()*[11]:

Listing 54-9
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setOptional(array('password'));
}
```

Options with defaults are already marked as optional.

When setting an option as optional, you can't be sure if it's in the array or not. You have to check if the option exists before using it.

To avoid checking if it exists everytime, you can also set a default of `null` to an option using the `setDefaults()` method (see Set Default Values), this means the element always exists in the array, but with a default of `null`.

Default Values that Depend on another Option

Suppose you add a `port` option to the `Mailer` class, whose default value you guess based on the encryption. You can do that easily by using a closure as the default value:

Listing 54-10
```php
use Symfony\Component\OptionsResolver\Options;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setDefaults(array(
        'encryption' => null,
```

9. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/Exception/MissingOptionsException.html
10. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#isRequired()
11. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setOptional()

```
        'port' => function (Options $options) {
            if ('ssl' === $options['encryption']) {
                return 465;
            }

            return 25;
        },
    ));
}
```

The *Options*[12] class implements *ArrayAccess*[13], *Iterator*[14] and *Countable*[15]. That means you can handle it just like a normal array containing the options.

The first argument of the closure must be typehinted as `Options`, otherwise it is considered as the value.

Overwriting default Values

A previously set default value can be overwritten by invoking *setDefaults()*[16] again. When using a closure as the new value it is passed 2 arguments:

- `$options`: an *Options*[17] instance with all the other default options
- `$previousValue`: the previous set default value

Listing 54-11

```
use Symfony\Component\OptionsResolver\Options;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...
    $resolver->setDefaults(array(
        'encryption' => 'ssl',
        'host' => 'localhost',
    ));

    // ...
    $resolver->setDefaults(array(
        'encryption' => 'tls', // simple overwrite
        'host' => function (Options $options, $previousValue) {
            return 'localhost' == $previousValue
                ? '127.0.0.1'
                : $previousValue;
        },
    ));
}
```

12. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/Options.html
13. http://php.net/manual/en/class.arrayaccess.php
14. http://php.net/manual/en/class.iterator.php
15. http://php.net/manual/en/class.countable.php
16. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setDefaults()
17. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/Options.html

If the previous default value is calculated by an expensive closure and you don't need access to it, you can use the *replaceDefaults()*[18] method instead. It acts like `setDefaults` but simply erases the previous value to improve performance. This means that the previous default value is not available when overwriting with another closure:

Listing 54-12
```php
use Symfony\Component\OptionsResolver\Options;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...
    $resolver->setDefaults(array(
        'encryption' => 'ssl',
        'heavy' => function (Options $options) {
            // Some heavy calculations to create the $result

            return $result;
        },
    ));

    $resolver->replaceDefaults(array(
        'encryption' => 'tls', // simple overwrite
        'heavy' => function (Options $options) {
            // $previousValue not available
            // ...

            return $someOtherResult;
        },
    ));
}
```

Existing option keys that you do not mention when overwriting are preserved.

Configure Allowed Values

Not all values are valid values for options. Suppose the `Mailer` class has a `transport` option, it can only be one of `sendmail`, `mail` or `smtp`. You can configure these allowed values by calling *setAllowedValues()*[19]:

Listing 54-13
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setAllowedValues(array(
        'encryption' => array(null, 'ssl', 'tls'),
    ));
}
```

18. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#replaceDefaults()
19. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setAllowedValues()

There is also an *addAllowedValues()*[20] method, which you can use if you want to add an allowed value to the previously configured allowed values.

Configure Allowed Types

You can also specify allowed types. For instance, the `port` option can be anything, but it must be an integer. You can configure these types by calling *setAllowedTypes()*[21]:

Listing 54-14
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setAllowedTypes(array(
        'port' => 'integer',
    ));
}
```

Possible types are the ones associated with the `is_*` PHP functions or a class name. You can also pass an array of types as the value. For instance, `array('null', 'string')` allows `port` to be `null` or a `string`.

There is also an *addAllowedTypes()*[22] method, which you can use to add an allowed type to the previous allowed types.

Normalize the Options

Some values need to be normalized before you can use them. For instance, pretend that the `host` should always start with `http://`. To do that, you can write normalizers. These closures will be executed after all options are passed and should return the normalized value. You can configure these normalizers by calling *setNormalizers()*[23]:

Listing 54-15
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
{
    // ...

    $resolver->setNormalizers(array(
        'host' => function (Options $options, $value) {
            if ('http://' !== substr($value, 0, 7)) {
                $value = 'http://'.$value;
            }

            return $value;
        },
    ));
}
```

You see that the closure also gets an `$options` parameter. Sometimes, you need to use the other options for normalizing:

Listing 54-16
```php
// ...
protected function setDefaultOptions(OptionsResolverInterface $resolver)
```

20. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#addAllowedValues()
21. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setAllowedTypes()
22. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#addAllowedTypes()
23. http://api.symfony.com/2.3/Symfony/Component/OptionsResolver/OptionsResolver.html#setNormalizers()

```
{
    // ...

    $resolver->setNormalizers(array(
        'host' => function (Options $options, $value) {
            if (!in_array(substr($value, 0, 7), array('http://', 'https://'))) {
                if ($options['ssl']) {
                    $value = 'https://'.$value;
                } else {
                    $value = 'http://'.$value;
                }
            }

            return $value;
        },
    ));
}
```

Chapter 55
The Process Component

> The Process component executes commands in sub-processes.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/process` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Process*[2]).

Usage

The ***Process***[3] class allows you to execute a command in a sub-process:

Listing 55-1
```php
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->run();

// executes after the command finishes
if (!$process->isSuccessful()) {
    throw new \RuntimeException($process->getErrorOutput());
}

print $process->getOutput();
```

The component takes care of the subtle differences between the different platforms when executing the command.

1. https://packagist.org/packages/symfony/process
2. https://github.com/symfony/Process
3. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html

New in version 2.2: The `getIncrementalOutput()` and `getIncrementalErrorOutput()` methods were introduced in Symfony 2.2.

The `getOutput()` method always returns the whole content of the standard output of the command and `getErrorOutput()` the content of the error output. Alternatively, the *`getIncrementalOutput()`*[4] and *`getIncrementalErrorOutput()`*[5] methods returns the new outputs since the last call.

Getting real-time Process Output

When executing a long running command (like rsync-ing files to a remote server), you can give feedback to the end user in real-time by passing an anonymous function to the *`run()`*[6] method:

Listing 55-2
```php
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->run(function ($type, $buffer) {
    if (Process::ERR === $type) {
        echo 'ERR > '.$buffer;
    } else {
        echo 'OUT > '.$buffer;
    }
});
```

New in version 2.1: The non-blocking feature was introduced in 2.1.

Running Processes Asynchronously

You can also start the subprocess and then let it run asynchronously, retrieving output and the status in your main process whenever you need it. Use the *`start()`*[7] method to start an asynchronous process, the *`isRunning()`*[8] method to check if the process is done and the *`getOutput()`*[9] method to get the output:

Listing 55-3
```php
$process = new Process('ls -lsa');
$process->start();

while ($process->isRunning()) {
    // waiting for process to finish
}

echo $process->getOutput();
```

You can also wait for a process to end if you started it asynchronously and are done doing other stuff:

Listing 55-4
```php
$process = new Process('ls -lsa');
$process->start();

// ... do other things

$process->wait(function ($type, $buffer) {
```

4. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#getIncrementalOutput()
5. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#getIncrementalErrorOutput()
6. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#run()
7. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#start()
8. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#isRunning()
9. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#getOutput()

```
7      if (Process::ERR === $type) {
8          echo 'ERR > '.$buffer;
9      } else {
10         echo 'OUT > '.$buffer;
11     }
12 });
```

The *wait()*[10] method is blocking, which means that your code will halt at this line until the external process is completed.

Stopping a Process

New in version 2.3: The `signal` parameter of the `stop` method was introduced in Symfony 2.3.

Any asynchronous process can be stopped at any time with the *stop()*[11] method. This method takes two arguments: a timeout and a signal. Once the timeout is reached, the signal is sent to the running process. The default signal sent to a process is `SIGKILL`. Please read the *signal documentation below* to find out more about signal handling in the Process component:

Listing 55-5
```
1 $process = new Process('ls -lsa');
2 $process->start();
3
4 // ... do other things
5
6 $process->stop(3, SIGINT);
```

Executing PHP Code in Isolation

If you want to execute some PHP code in isolation, use the `PhpProcess` instead:

Listing 55-6
```
1 use Symfony\Component\Process\PhpProcess;
2
3 $process = new PhpProcess(<<<EOF
4     <?php echo 'Hello World'; ?>
5 EOF
6 );
7 $process->run();
```

To make your code work better on all platforms, you might want to use the *ProcessBuilder*[12] class instead:

Listing 55-7
```
1 use Symfony\Component\Process\ProcessBuilder;
2
3 $builder = new ProcessBuilder(array('ls', '-lsa'));
4 $builder->getProcess()->run();
```

10. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#wait()
11. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#stop()
12. http://api.symfony.com/2.3/Symfony/Component/Process/ProcessBuilder.html

New in version 2.3: The *`ProcessBuilder::setPrefix`*[13] method was introduced in Symfony 2.3.

In case you are building a binary driver, you can use the *`setPrefix()`*[14] method to prefix all the generated process commands.

The following example will generate two process commands for a tar binary adapter:

Listing 55-8
```php
use Symfony\Component\Process\ProcessBuilder;

$builder = new ProcessBuilder();
$builder->setPrefix('/usr/bin/tar');

// '/usr/bin/tar' '--list' '--file=archive.tar.gz'
echo $builder
    ->setArguments(array('--list', '--file=archive.tar.gz'))
    ->getProcess()
    ->getCommandLine();

// '/usr/bin/tar' '-xzf' 'archive.tar.gz'
echo $builder
    ->setArguments(array('-xzf', 'archive.tar.gz'))
    ->getProcess()
    ->getCommandLine();
```

Process Timeout

You can limit the amount of time a process takes to complete by setting a timeout (in seconds):

Listing 55-9
```php
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->setTimeout(3600);
$process->run();
```

If the timeout is reached, a *`RuntimeException`*[15] is thrown.

For long running commands, it is your responsibility to perform the timeout check regularly:

Listing 55-10
```php
$process->setTimeout(3600);
$process->start();

while ($condition) {
    // ...

    // check if the timeout is reached
    $process->checkTimeout();

    usleep(200000);
}
```

13. http://api.symfony.com/2.3/Symfony/Component/Process/ProcessBuilder.html#setPrefix()
14. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#setPrefix()
15. http://api.symfony.com/2.3/Symfony/Process/Exception/RuntimeException.html

Process Signals

New in version 2.3: The `signal` method was introduced in Symfony 2.3.

When running a program asynchronously, you can send it POSIX signals with the *`signal()`*[16] method:

Listing 55-11
```php
use Symfony\Component\Process\Process;

$process = new Process('find / -name "rabbit"');
$process->start();

// will send a SIGKILL to the process
$process->signal(SIGKILL);
```

Due to some limitations in PHP, if you're using signals with the Process component, you may have to prefix your commands with *exec*[17]. Please read *Symfony Issue#5759*[18] and *PHP Bug#39992*[19] to understand why this is happening.

POSIX signals are not available on Windows platforms, please refer to the *PHP documentation*[20] for available signals.

Process Pid

New in version 2.3: The `getPid` method was introduced in Symfony 2.3.

You can access the *pid*[21] of a running process with the *`getPid()`*[22] method.

Listing 55-12
```php
use Symfony\Component\Process\Process;

$process = new Process('/usr/bin/php worker.php');
$process->start();

$pid = $process->getPid();
```

Due to some limitations in PHP, if you want to get the pid of a symfony Process, you may have to prefix your commands with *exec*[23]. Please read *Symfony Issue#5759*[24] to understand why this is happening.

16. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#signal()
17. http://en.wikipedia.org/wiki/Exec_(operating_system)
18. https://github.com/symfony/symfony/issues/5759
19. https://bugs.php.net/bug.php?id=39992
20. http://php.net/manual/en/pcntl.constants.php
21. http://en.wikipedia.org/wiki/Process_identifier
22. http://api.symfony.com/2.3/Symfony/Component/Process/Process.html#getPid()
23. http://en.wikipedia.org/wiki/Exec_(operating_system)
24. https://github.com/symfony/symfony/issues/5759

Chapter 56
The PropertyAccess Component

> The PropertyAccess component provides function to read and write from/to an object or array using a simple string notation.

New in version 2.2: The PropertyAccess component was introduced in Symfony 2.2. Previously, the `PropertyPath` class was located in the Form component.

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/property-access` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/PropertyAccess*[2]).

Usage

The entry point of this component is the *`PropertyAccess::createPropertyAccessor`*[3] factory. This factory will create a new instance of the *`PropertyAccessor`*[4] class with the default configuration:

Listing 56-1
```php
use Symfony\Component\PropertyAccess\PropertyAccess;

$accessor = PropertyAccess::createPropertyAccessor();
```

New in version 2.3: The *`createPropertyAccessor()`*[5] method was introduced in Symfony 2.3. Previously, it was called `getPropertyAccessor()`.

1. https://packagist.org/packages/symfony/property-access
2. https://github.com/symfony/PropertyAccess
3. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccess.html#createPropertyAccessor()
4. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessor.html
5. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccess.html#createPropertyAccessor()

Reading from Arrays

You can read an array with the *PropertyAccessor::getValue*[6] method. This is done using the index notation that is used in PHP:

Listing 56-2
```php
// ...
$person = array(
    'first_name' => 'Wouter',
);

echo $accessor->getValue($person, '[first_name]'); // 'Wouter'
echo $accessor->getValue($person, '[age]'); // null
```

As you can see, the method will return `null` if the index does not exists.

You can also use multi dimensional arrays:

Listing 56-3
```php
// ...
$persons = array(
    array(
        'first_name' => 'Wouter',
    ),
    array(
        'first_name' => 'Ryan',
    )
);

echo $accessor->getValue($persons, '[0][first_name]'); // 'Wouter'
echo $accessor->getValue($persons, '[1][first_name]'); // 'Ryan'
```

Reading from Objects

The `getValue` method is a very robust method, and you can see all of its features when working with objects.

Accessing public Properties

To read from properties, use the "dot" notation:

Listing 56-4
```php
// ...
$person = new Person();
$person->firstName = 'Wouter';

echo $accessor->getValue($person, 'firstName'); // 'Wouter'

$child = new Person();
$child->firstName = 'Bar';
$person->children = array($child);

echo $accessor->getValue($person, 'children[0].firstName'); // 'Bar'
```

6. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessor.html#getValue()

Accessing public properties is the last option used by `PropertyAccessor`. It tries to access the value using the below methods first before using the property directly. For example, if you have a public property that has a getter method, it will use the getter.

Using Getters

The `getValue` method also supports reading using getters. The method will be created using common naming conventions for getters. It camelizes the property name (`first_name` becomes `FirstName`) and prefixes it with `get`. So the actual method becomes `getFirstName`:

Listing 56-5

```
// ...
class Person
{
    private $firstName = 'Wouter';

    public function getFirstName()
    {
        return $this->firstName;
    }
}

$person = new Person();

echo $accessor->getValue($person, 'first_name'); // 'Wouter'
```

Using Hassers/Issers

And it doesn't even stop there. If there is no getter found, the accessor will look for an isser or hasser. This method is created using the same way as getters, this means that you can do something like this:

Listing 56-6

```
// ...
class Person
{
    private $author = true;
    private $children = array();

    public function isAuthor()
    {
        return $this->author;
    }

    public function hasChildren()
    {
        return 0 !== count($this->children);
    }
}

$person = new Person();

if ($accessor->getValue($person, 'author')) {
    echo 'He is an author';
}
if ($accessor->getValue($person, 'children')) {
    echo 'He has children';
}
```

This will produce: `He is an author`

Magic `__get()` Method

The `getValue` method can also use the magic `__get` method:

Listing 56-7
```php
// ...
class Person
{
    private $children = array(
        'Wouter' => array(...),
    );

    public function __get($id)
    {
        return $this->children[$id];
    }
}

$person = new Person();

echo $accessor->getValue($person, 'Wouter'); // array(...)
```

Magic `__call()` Method

At last, `getValue` can use the magic `__call` method, but you need to enable this feature by using *`PropertyAccessorBuilder`*[7]:

Listing 56-8
```php
// ...
class Person
{
    private $children = array(
        'wouter' => array(...),
    );

    public function __call($name, $args)
    {
        $property = lcfirst(substr($name, 3));
        if ('get' === substr($name, 0, 3)) {
            return isset($this->children[$property])
                ? $this->children[$property]
                : null;
        } elseif ('set' === substr($name, 0, 3)) {
            $value = 1 == count($args) ? $args[0] : null;
            $this->children[$property] = $value;
        }
    }
}

$person = new Person();

// Enable magic __call
$accessor = PropertyAccess::createPropertyAccessorBuilder()
    ->enableMagicCall()
    ->getPropertyAccessor();
```

7. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html

```
28
29 echo $accessor->getValue($person, 'wouter'); // array(...)
```

New in version 2.3: The use of magic `__call()` method was introduced in Symfony 2.3.

The `__call` feature is disabled by default, you can enable it by calling *`PropertyAccessorBuilder::enableMagicCallEnabled`*[8] see Enable other Features.

Writing to Arrays

The `PropertyAccessor` class can do more than just read an array, it can also write to an array. This can be achieved using the *`PropertyAccessor::setValue`*[9] method:

Listing 56-9
```
1 // ...
2 $person = array();
3
4 $accessor->setValue($person, '[first_name]', 'Wouter');
5
6 echo $accessor->getValue($person, '[first_name]'); // 'Wouter'
7 // or
8 // echo $person['first_name']; // 'Wouter'
```

Writing to Objects

The `setValue` method has the same features as the `getValue` method. You can use setters, the magic `__set` method or properties to set values:

Listing 56-10
```
1  // ...
2  class Person
3  {
4      public $firstName;
5      private $lastName;
6      private $children = array();
7
8      public function setLastName($name)
9      {
10         $this->lastName = $name;
11     }
12
13     public function __set($property, $value)
14     {
15         $this->$property = $value;
16     }
17
18     // ...
19 }
20
```

8. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html#enableMagicCallEnabled()
9. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessor.html#setValue()

```php
$person = new Person();

$accessor->setValue($person, 'firstName', 'Wouter');
$accessor->setValue($person, 'lastName', 'de Jong');
$accessor->setValue($person, 'children', array(new Person()));

echo $person->firstName; // 'Wouter'
echo $person->getLastName(); // 'de Jong'
echo $person->children; // array(Person());
```

You can also use `__call` to set values but you need to enable the feature, see Enable other Features.

Listing 56-11
```php
// ...
class Person
{
    private $children = array();

    public function __call($name, $args)
    {
        $property = lcfirst(substr($name, 3));
        if ('get' === substr($name, 0, 3)) {
            return isset($this->children[$property])
                ? $this->children[$property]
                : null;
        } elseif ('set' === substr($name, 0, 3)) {
            $value = 1 == count($args) ? $args[0] : null;
            $this->children[$property] = $value;
        }
    }

}

$person = new Person();

// Enable magic __call
$accessor = PropertyAccess::createPropertyAccessorBuilder()
    ->enableMagicCall()
    ->getPropertyAccessor();

$accessor->setValue($person, 'wouter', array(...));

echo $person->getWouter(); // array(...)
```

Mixing Objects and Arrays

You can also mix objects and arrays:

Listing 56-12
```php
// ...
class Person
{
    public $firstName;
    private $children = array();

    public function setChildren($children)
    {
```

```
        $this->children = $children;
    }

    public function getChildren()
    {
        return $this->children;
    }
}

$person = new Person();

$accessor->setValue($person, 'children[0]', new Person);
// equal to $person->getChildren()[0] = new Person()

$accessor->setValue($person, 'children[0].firstName', 'Wouter');
// equal to $person->getChildren()[0]->firstName = 'Wouter'

echo 'Hello '.$accessor->getValue($person, 'children[0].firstName'); // 'Wouter'
// equal to $person->getChildren()[0]->firstName
```

Enable other Features

The *PropertyAccessor*[10] can be configured to enable extra features. To do that you could use the *PropertyAccessorBuilder*[11]:

Listing 56-13
```
// ...
$accessorBuilder = PropertyAccess::createPropertyAccessorBuilder();

// Enable magic __call
$accessorBuilder->enableMagicCall();

// Disable magic __call
$accessorBuilder->disableMagicCall();

// Check if magic __call handling is enabled
$accessorBuilder->isMagicCallEnabled(); // true or false

// At the end get the configured property accessor
$accessor = $accessorBuilder->getPropertyAccessor();

// Or all in one
$accessor = PropertyAccess::createPropertyAccessorBuilder()
    ->enableMagicCall()
    ->getPropertyAccessor();
```

Or you can pass parameters directly to the constructor (not the recommended way):

Listing 56-14
```
// ...
$accessor = new PropertyAccessor(true); // this enables handling of magic __call
```

10. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessor.html
11. http://api.symfony.com/2.3/Symfony/Component/PropertyAccess/PropertyAccessorBuilder.html

Chapter 57
The Routing Component

> The Routing component maps an HTTP request to a set of configuration variables.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/routing` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Routing*[2]).

Usage

In order to set up a basic routing system you need three parts:

- A *`RouteCollection`*[3], which contains the route definitions (instances of the class *`Route`*[4])
- A *`RequestContext`*[5], which has information about the request
- A *`UrlMatcher`*[6], which performs the mapping of the request to a single route

Here is a quick example. Notice that this assumes that you've already configured your autoloader to load the Routing component:

Listing 57-1
```
1 use Symfony\Component\Routing\Matcher\UrlMatcher;
2 use Symfony\Component\Routing\RequestContext;
3 use Symfony\Component\Routing\RouteCollection;
4 use Symfony\Component\Routing\Route;
```

1. https://packagist.org/packages/symfony/routing
2. https://github.com/symfony/Routing
3. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html
4. http://api.symfony.com/2.3/Symfony/Component/Routing/Route.html
5. http://api.symfony.com/2.3/Symfony/Component/Routing/RequestContext.html
6. http://api.symfony.com/2.3/Symfony/Component/Routing/Matcher/UrlMatcher.html

```php

$route = new Route('/foo', array('controller' => 'MyController'));
$routes = new RouteCollection();
$routes->add('route_name', $route);

$context = new RequestContext($_SERVER['REQUEST_URI']);

$matcher = new UrlMatcher($routes, $context);

$parameters = $matcher->match('/foo');
// array('controller' => 'MyController', '_route' => 'route_name')
```

Be careful when using `$_SERVER['REQUEST_URI']`, as it may include any query parameters on the URL, which will cause problems with route matching. An easy way to solve this is to use the HttpFoundation component as explained *below*.

You can add as many routes as you like to a *`RouteCollection`*[7].

The *`RouteCollection::add()`*[8] method takes two arguments. The first is the name of the route. The second is a *`Route`*[9] object, which expects a URL path and some array of custom variables in its constructor. This array of custom variables can be *anything* that's significant to your application, and is returned when that route is matched.

If no matching route can be found a *`ResourceNotFoundException`*[10] will be thrown.

In addition to your array of custom variables, a `_route` key is added, which holds the name of the matched route.

Defining Routes

A full route definition can contain up to seven parts:

1. The URL path route. This is matched against the URL passed to the *RequestContext*, and can contain named wildcard placeholders (e.g. `{placeholders}`) to match dynamic parts in the URL.
2. An array of default values. This contains an array of arbitrary values that will be returned when the request matches the route.
3. An array of requirements. These define constraints for the values of the placeholders as regular expressions.
4. An array of options. These contain internal settings for the route and are the least commonly needed.
5. A host. This is matched against the host of the request. See *How to Match a Route Based on the Host* for more details.
6. An array of schemes. These enforce a certain HTTP scheme (`http`, `https`).
7. An array of methods. These enforce a certain HTTP request method (`HEAD`, `GET`, `POST`, ...).

New in version 2.2: Host matching support was introduced in Symfony 2.2

Take the following route, which combines several of these ideas:

Listing 57-2
```php
$route = new Route(
    '/archive/{month}', // path
```

7. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html
8. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html#add()
9. http://api.symfony.com/2.3/Symfony/Component/Routing/Route.html
10. http://api.symfony.com/2.3/Symfony/Component/Routing/Exception/ResourceNotFoundException.html

```
    array('controller' => 'showArchive'), // default values
    array('month' => '[0-9]{4}-[0-9]{2}', 'subdomain' => 'www|m'), // requirements
    array(), // options
    '{subdomain}.example.com', // host
    array(), // schemes
    array() // methods
);

// ...

$parameters = $matcher->match('/archive/2012-01');
// array(
//     'controller' => 'showArchive',
//     'month' => '2012-01',
//     'subdomain' => 'www',
//     '_route' => ...
// )

$parameters = $matcher->match('/archive/foo');
// throws ResourceNotFoundException
```

In this case, the route is matched by `/archive/2012-01`, because the `{month}` wildcard matches the regular expression wildcard given. However, `/archive/foo` does *not* match, because "foo" fails the month wildcard.

If you want to match all URLs which start with a certain path and end in an arbitrary suffix you can use the following route definition:

Listing 57-3
```
$route = new Route(
    '/start/{suffix}',
    array('suffix' => ''),
    array('suffix' => '.*')
);
```

Using Prefixes

You can add routes or other instances of *`RouteCollection`*[11] to *another* collection. This way you can build a tree of routes. Additionally you can define a prefix and default values for the parameters, requirements, options, schemes and the host to all routes of a subtree using methods provided by the `RouteCollection` class:

Listing 57-4
```
$rootCollection = new RouteCollection();

$subCollection = new RouteCollection();
$subCollection->add(...);
$subCollection->add(...);
$subCollection->addPrefix('/prefix');
$subCollection->addDefaults(array(...));
$subCollection->addRequirements(array(...));
$subCollection->addOptions(array(...));
$subCollection->setHost('admin.example.com');
$subCollection->setMethods(array('POST'));
```

11. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html

```
12 $subCollection->setSchemes(array('https'));
13
14 $rootCollection->addCollection($subCollection);
```

Set the Request Parameters

The *RequestContext*[12] provides information about the current request. You can define all parameters of an HTTP request with this class via its constructor:

Listing 57-5
```
1  public function __construct(
2      $baseUrl = '',
3      $method = 'GET',
4      $host = 'localhost',
5      $scheme = 'http',
6      $httpPort = 80,
7      $httpsPort = 443,
8      $path = '/',
9      $queryString = ''
10 )
```

Normally you can pass the values from the `$_SERVER` variable to populate the *RequestContext*[13]. But If you use the *HttpFoundation* component, you can use its *Request*[14] class to feed the *RequestContext*[15] in a shortcut:

Listing 57-6
```
1 use Symfony\Component\HttpFoundation\Request;
2
3 $context = new RequestContext();
4 $context->fromRequest(Request::createFromGlobals());
```

Generate a URL

While the *UrlMatcher*[16] tries to find a route that fits the given request you can also build a URL from a certain route:

Listing 57-7
```
1  use Symfony\Component\Routing\Generator\UrlGenerator;
2
3  $routes = new RouteCollection();
4  $routes->add('show_post', new Route('/show/{slug}'));
5
6  $context = new RequestContext($_SERVER['REQUEST_URI']);
7
8  $generator = new UrlGenerator($routes, $context);
9
10 $url = $generator->generate('show_post', array(
11     'slug' => 'my-blog-post',
12 ));
13 // /show/my-blog-post
```

12. http://api.symfony.com/2.3/Symfony/Component/Routing/RequestContext.html
13. http://api.symfony.com/2.3/Symfony/Component/Routing/RequestContext.html
14. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
15. http://api.symfony.com/2.3/Symfony/Component/Routing/RequestContext.html
16. http://api.symfony.com/2.3/Symfony/Component/Routing/Matcher/UrlMatcher.html

If you have defined a scheme, an absolute URL is generated if the scheme of the current *RequestContext*[17] does not match the requirement.

Load Routes from a File

You've already seen how you can easily add routes to a collection right inside PHP. But you can also load routes from a number of different files.

The Routing component comes with a number of loader classes, each giving you the ability to load a collection of route definitions from an external file of some format. Each loader expects a *FileLocator*[18] instance as the constructor argument. You can use the *FileLocator*[19] to define an array of paths in which the loader will look for the requested files. If the file is found, the loader returns a *RouteCollection*[20].

If you're using the `YamlFileLoader`, then route definitions look like this:

Listing 57-8
```yaml
# routes.yml
route1:
    path:     /foo
    defaults: { _controller: 'MyController::fooAction' }

route2:
    path:     /foo/bar
    defaults: { _controller: 'MyController::foobarAction' }
```

To load this file, you can use the following code. This assumes that your `routes.yml` file is in the same directory as the below code:

Listing 57-9
```php
use Symfony\Component\Config\FileLocator;
use Symfony\Component\Routing\Loader\YamlFileLoader;

// look inside *this* directory
$locator = new FileLocator(array(__DIR__));
$loader = new YamlFileLoader($locator);
$collection = $loader->load('routes.yml');
```

Besides *YamlFileLoader*[21] there are two other loaders that work the same way:

- *XmlFileLoader*[22]
- *PhpFileLoader*[23]

If you use the *PhpFileLoader*[24] you have to provide the name of a PHP file which returns a *RouteCollection*[25]:

Listing 57-10
```php
// RouteProvider.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
```

17. http://api.symfony.com/2.3/Symfony/Component/Routing/RequestContext.html
18. http://api.symfony.com/2.3/Symfony/Component/Config/FileLocator.html
19. http://api.symfony.com/2.3/Symfony/Component/Config/FileLocator.html
20. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html
21. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/YamlFileLoader.html
22. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/XmlFileLoader.html
23. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/PhpFileLoader.html
24. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/PhpFileLoader.html
25. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html

```
4
5  $collection = new RouteCollection();
6  $collection->add(
7      'route_name',
8      new Route('/foo', array('controller' => 'ExampleController'))
9  );
10 // ...
11
12 return $collection;
```

Routes as Closures

There is also the *ClosureLoader*[26], which calls a closure and uses the result as a *RouteCollection*[27]:

Listing 57-11
```
1  use Symfony\Component\Routing\Loader\ClosureLoader;
2
3  $closure = function () {
4      return new RouteCollection();
5  };
6
7  $loader = new ClosureLoader();
8  $collection = $loader->load($closure);
```

Routes as Annotations

Last but not least there are *AnnotationDirectoryLoader*[28] and *AnnotationFileLoader*[29] to load route definitions from class annotations. The specific details are left out here.

The all-in-one Router

The *Router*[30] class is an all-in-one package to quickly use the Routing component. The constructor expects a loader instance, a path to the main route definition and some other settings:

Listing 57-12
```
1  public function __construct(
2      LoaderInterface $loader,
3      $resource,
4      array $options = array(),
5      RequestContext $context = null,
6      array $defaults = array()
7  );
```

With the `cache_dir` option you can enable route caching (if you provide a path) or disable caching (if it's set to `null`). The caching is done automatically in the background if you want to use it. A basic example of the *Router*[31] class would look like:

Listing 57-13
```
1  $locator = new FileLocator(array(__DIR__));
2  $requestContext = new RequestContext($_SERVER['REQUEST_URI']);
3
```

26. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/ClosureLoader.html
27. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html
28. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/AnnotationDirectoryLoader.html
29. http://api.symfony.com/2.3/Symfony/Component/Routing/Loader/AnnotationFileLoader.html
30. http://api.symfony.com/2.3/Symfony/Component/Routing/Router.html
31. http://api.symfony.com/2.3/Symfony/Component/Routing/Router.html

```
$router = new Router(
    new YamlFileLoader($locator),
    'routes.yml',
    array('cache_dir' => __DIR__.'/cache'),
    $requestContext
);
$router->match('/foo/bar');
```

If you use caching, the Routing component will compile new classes which are saved in the `cache_dir`. This means your script must have write permissions for that location.

Chapter 58

How to Match a Route Based on the Host

New in version 2.2: Host matching support was introduced in Symfony 2.2

You can also match on the HTTP *host* of the incoming request.

Listing 58-1

```yaml
mobile_homepage:
    path:     /
    host:     m.example.com
    defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }

homepage:
    path:     /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Both routes match the same path `/`, however the first one will match only if the host is `m.example.com`.

Using Placeholders

The host option uses the same syntax as the path matching system. This means you can use placeholders in your hostname:

Listing 58-2

```yaml
projects_homepage:
    path:     /
    host:     "{project_name}.example.com"
    defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }

homepage:
    path:     /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

You can also set requirements and default options for these placeholders. For instance, if you want to match both `m.example.com` and `mobile.example.com`, you use this:

Listing 58-3

```yaml
mobile_homepage:
    path:     /
    host:     "{subdomain}.example.com"
    defaults:
        _controller: AcmeDemoBundle:Main:mobileHomepage
        subdomain: m
    requirements:
        subdomain: m|mobile

homepage:
    path:     /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

You can also use service parameters if you do not want to hardcode the hostname:

Listing 58-4
```yaml
mobile_homepage:
    path:     /
    host:     "m.{domain}"
    defaults:
        _controller: AcmeDemoBundle:Main:mobileHomepage
        domain: "%domain%"
    requirements:
        domain: "%domain%"

homepage:
    path:  /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Make sure you also include a default option for the `domain` placeholder, otherwise you need to include a domain value each time you generate a URL using the route.

Using Host Matching of Imported Routes

You can also set the host option on imported routes:

Listing 58-5
```yaml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
    host:     "hello.example.com"
```

The host `hello.example.com` will be set on each route loaded from the new routing resource.

Testing your Controllers

You need to set the Host HTTP header on your request objects if you want to get past url matching in your functional tests.

Listing 58-6
```php
$crawler = $client->request(
    'GET',
```

```
    '/homepage',
    array(),
    array(),
    array('HTTP_HOST' => 'm.' . $client->getContainer()->getParameter('domain'))
);
```

Chapter 59

The Security Component

> The Security component provides a complete security system for your web application. It ships with facilities for authenticating using HTTP basic or digest authentication, interactive form login or X.509 certificate login, but also allows you to implement your own authentication strategies. Furthermore, the component provides ways to authorize authenticated users based on their roles, and it contains an advanced ACL system.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/security` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Security*[2]).

Sections

- *The Firewall and Security Context*
- *Authentication*
- *Authorization*
- *Securely Comparing Strings and Generating Random Numbers*

1. `https://packagist.org/packages/symfony/security`
2. `https://github.com/symfony/Security`

Chapter 60
The Firewall and Security Context

Central to the Security component is the security context, which is an instance of *`SecurityContextInterface`*[1]. When all steps in the process of authenticating the user have been taken successfully, you can ask the security context if the authenticated user has access to a certain action or resource of the application:

Listing 60-1
```php
use Symfony\Component\Security\Core\SecurityContext;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// instance of
Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface
$authenticationManager = ...;

// instance of Symfony\Component\Security\Core\Authorization\AccessDecisionManagerInterface
$accessDecisionManager = ...;

$securityContext = new SecurityContext(
    $authenticationManager,
    $accessDecisionManager
);

// ... authenticate the user

if (!$securityContext->isGranted('ROLE_ADMIN')) {
    throw new AccessDeniedException();
}
```

Read the dedicated sections to learn more about *Authentication* and *Authorization*.

1. http://api.symfony.com/2.3/Symfony/Component/Security/Core/SecurityContextInterface.html

A Firewall for HTTP Requests

Authenticating a user is done by the firewall. An application may have multiple secured areas, so the firewall is configured using a map of these secured areas. For each of these areas, the map contains a request matcher and a collection of listeners. The request matcher gives the firewall the ability to find out if the current request points to a secured area. The listeners are then asked if the current request can be used to authenticate the user:

Listing 60-2
```php
use Symfony\Component\Security\Http\FirewallMap;
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\ExceptionListener;

$map = new FirewallMap();

$requestMatcher = new RequestMatcher('^/secured-area/');

// instances of Symfony\Component\Security\Http\Firewall\ListenerInterface
$listeners = array(...);

$exceptionListener = new ExceptionListener(...);

$map->add($requestMatcher, $listeners, $exceptionListener);
```

The firewall map will be given to the firewall as its first argument, together with the event dispatcher that is used by the *HttpKernel*[2]:

Listing 60-3
```php
use Symfony\Component\Security\Http\Firewall;
use Symfony\Component\HttpKernel\KernelEvents;

// the EventDispatcher used by the HttpKernel
$dispatcher = ...;

$firewall = new Firewall($map, $dispatcher);

$dispatcher->addListener(
    KernelEvents::REQUEST,
    array($firewall, 'onKernelRequest')
);
```

The firewall is registered to listen to the `kernel.request` event that will be dispatched by the HttpKernel at the beginning of each request it processes. This way, the firewall may prevent the user from going any further than allowed.

Firewall Listeners

When the firewall gets notified of the `kernel.request` event, it asks the firewall map if the request matches one of the secured areas. The first secured area that matches the request will return a set of corresponding firewall listeners (which each implement *ListenerInterface*[3]). These listeners will all be asked to handle the current request. This basically means: find out if the current request contains any information by which the user might be authenticated (for instance the Basic HTTP authentication listener checks if the request has a header called `PHP_AUTH_USER`).

2. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernel.html
3. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall/ListenerInterface.html

Exception Listener

If any of the listeners throws an *AuthenticationException*[4], the exception listener that was provided when adding secured areas to the firewall map will jump in.

The exception listener determines what happens next, based on the arguments it received when it was created. It may start the authentication procedure, perhaps ask the user to supply their credentials again (when they have only been authenticated based on a "remember-me" cookie), or transform the exception into an *AccessDeniedHttpException*[5], which will eventually result in an "HTTP/1.1 403: Access Denied" response.

Entry Points

When the user is not authenticated at all (i.e. when the security context has no token yet), the firewall's entry point will be called to "start" the authentication process. An entry point should implement *AuthenticationEntryPointInterface*[6], which has only one method: *start()*[7]. This method receives the current *Request*[8] object and the exception by which the exception listener was triggered. The method should return a *Response*[9] object. This could be, for instance, the page containing the login form or, in the case of Basic HTTP authentication, a response with a `WWW-Authenticate` header, which will prompt the user to supply their username and password.

Flow: Firewall, Authentication, Authorization

Hopefully you can now see a little bit about how the "flow" of the security context works:

1. The Firewall is registered as a listener on the `kernel.request` event;
2. At the beginning of the request, the Firewall checks the firewall map to see if any firewall should be active for this URL;
3. If a firewall is found in the map for this URL, its listeners are notified;
4. Each listener checks to see if the current request contains any authentication information - a listener may (a) authenticate a user, (b) throw an `AuthenticationException`, or (c) do nothing (because there is no authentication information on the request);
5. Once a user is authenticated, you'll use *Authorization* to deny access to certain resources.

Read the next sections to find out more about *Authentication* and *Authorization*.

4. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Exception/AuthenticationException.html
5. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Exception/AccessDeniedHttpException.html
6. http://api.symfony.com/2.3/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html
7. http://api.symfony.com/2.3/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html#start()
8. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
9. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Response.html

Chapter 61
Authentication

When a request points to a secured area, and one of the listeners from the firewall map is able to extract the user's credentials from the current *Request*[1] object, it should create a token, containing these credentials. The next thing the listener should do is ask the authentication manager to validate the given token, and return an *authenticated* token if the supplied credentials were found to be valid. The listener should then store the authenticated token in the security context:

Listing 61-1
```php
use Symfony\Component\Security\Http\Firewall\ListenerInterface;
use Symfony\Component\Security\Core\SecurityContextInterface;
use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;

class SomeAuthenticationListener implements ListenerInterface
{
    /**
     * @var SecurityContextInterface
     */
    private $securityContext;

    /**
     * @var AuthenticationManagerInterface
     */
    private $authenticationManager;

    /**
     * @var string Uniquely identifies the secured area
     */
    private $providerKey;

    // ...

    public function handle(GetResponseEvent $event)
    {
        $request = $event->getRequest();
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html

```
29
30          $username = ...;
31          $password = ...;
32
33          $unauthenticatedToken = new UsernamePasswordToken(
34              $username,
35              $password,
36              $this->providerKey
37          );
38
39          $authenticatedToken = $this
40              ->authenticationManager
41              ->authenticate($unauthenticatedToken);
42
43          $this->securityContext->setToken($authenticatedToken);
44      }
45  }
```

A token can be of any class, as long as it implements *TokenInterface*[2].

The Authentication Manager

The default authentication manager is an instance of *AuthenticationProviderManager*[3]:

Listing 61-2

```
1  use Symfony\Component\Security\Core\Authentication\AuthenticationProviderManager;
2
3  // instances of
4  Symfony\Component\Security\Core\Authentication\AuthenticationProviderInterface
5  $providers = array(...);
6
7  $authenticationManager = new AuthenticationProviderManager($providers);
8
9  try {
10     $authenticatedToken = $authenticationManager
11         ->authenticate($unauthenticatedToken);
12 } catch (AuthenticationException $failed) {
13     // authentication failed
   }
```

The `AuthenticationProviderManager`, when instantiated, receives several authentication providers, each supporting a different type of token.

You may of course write your own authentication manager, it only has to implement *AuthenticationManagerInterface*[4].

2. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html
3. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/AuthenticationProviderManager.html
4. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/AuthenticationManagerInterface.html

Authentication Providers

Each provider (since it implements *AuthenticationProviderInterface*[5]) has a method *supports()*[6] by which the `AuthenticationProviderManager` can determine if it supports the given token. If this is the case, the manager then calls the provider's method *AuthenticationProviderInterface::authenticate*[7]. This method should return an authenticated token or throw an *AuthenticationException*[8] (or any other exception extending it).

Authenticating Users by their Username and Password

An authentication provider will attempt to authenticate a user based on the credentials they provided. Usually these are a username and a password. Most web applications store their user's username and a hash of the user's password combined with a randomly generated salt. This means that the average authentication would consist of fetching the salt and the hashed password from the user data storage, hash the password the user has just provided (e.g. using a login form) with the salt and compare both to determine if the given password is valid.

This functionality is offered by the *DaoAuthenticationProvider*[9]. It fetches the user's data from a *UserProviderInterface*[10], uses a *PasswordEncoderInterface*[11] to create a hash of the password and returns an authenticated token if the password was valid:

Listing 61-3

```
use Symfony\Component\Security\Core\Authentication\Provider\DaoAuthenticationProvider;
use Symfony\Component\Security\Core\User\UserChecker;
use Symfony\Component\Security\Core\User\InMemoryUserProvider;
use Symfony\Component\Security\Core\Encoder\EncoderFactory;

$userProvider = new InMemoryUserProvider(
    array(
        'admin' => array(
            // password is "foo"
            'password' =>
'5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg==',
            'roles'    => array('ROLE_ADMIN'),
        ),
    )
);

// for some extra checks: is account enabled, locked, expired, etc.?
$userChecker = new UserChecker();

// an array of password encoders (see below)
$encoderFactory = new EncoderFactory(...);

$provider = new DaoAuthenticationProvider(
    $userProvider,
    $userChecker,
    'secured_area',
    $encoderFactory
);
```

5. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html
6. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html#supports()
7. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface::authenticate.html
8. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Exception/AuthenticationException.html
9. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
10. http://api.symfony.com/2.3/Symfony/Component/Security/Core/User/UserProviderInterface.html
11. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
28
29 $provider->authenticate($unauthenticatedToken);
```

The example above demonstrates the use of the "in-memory" user provider, but you may use any user provider, as long as it implements *UserProviderInterface*[12]. It is also possible to let multiple user providers try to find the user's data, using the *ChainUserProvider*[13].

The Password Encoder Factory

The *DaoAuthenticationProvider*[14] uses an encoder factory to create a password encoder for a given type of user. This allows you to use different encoding strategies for different types of users. The default *EncoderFactory*[15] receives an array of encoders:

Listing 61-4

```php
use Symfony\Component\Security\Core\Encoder\EncoderFactory;
use Symfony\Component\Security\Core\Encoder\MessageDigestPasswordEncoder;

$defaultEncoder = new MessageDigestPasswordEncoder('sha512', true, 5000);
$weakEncoder = new MessageDigestPasswordEncoder('md5', true, 1);

$encoders = array(
    'Symfony\\Component\\Security\\Core\\User\\User' => $defaultEncoder,
    'Acme\\Entity\\LegacyUser'                       => $weakEncoder,

    // ...
);

$encoderFactory = new EncoderFactory($encoders);
```

Each encoder should implement *PasswordEncoderInterface*[16] or be an array with a `class` and an `arguments` key, which allows the encoder factory to construct the encoder only when it is needed.

Creating a custom Password Encoder

There are many built-in password encoders. But if you need to create your own, it just needs to follow these rules:

1. The class must implement *PasswordEncoderInterface*[17];
2. The implementations of *encodePassword()*[18] and *isPasswordValid()*[19] must first of all make sure the password is not too long, i.e. the password length is no longer than 4096 characters. This is for security reasons (see *CVE-2013-5750*[20]), and you can use the *isPasswordTooLong()*[21] method for this check:

Listing 61-5

12. http://api.symfony.com/2.3/Symfony/Component/Security/Core/User/UserProviderInterface.html
13. http://api.symfony.com/2.3/Symfony/Component/Security/Core/User/ChainUserProvider.html
14. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
15. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/EncoderFactory.html
16. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html
17. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html
18. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html#encodePassword()
19. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html#isPasswordValid()
20. http://symfony.com/blog/cve-2013-5750-security-issue-in-fosuserbundle-login-form
21. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/BasePasswordEncoder.html#isPasswordTooLong()

```php
use Symfony\Component\Security\Core\Exception\BadCredentialsException;

class FoobarEncoder extends BasePasswordEncoder
{
    public function encodePassword($raw, $salt)
    {
        if ($this->isPasswordTooLong($raw)) {
            throw new BadCredentialsException('Invalid password.');
        }

        // ...
    }

    public function isPasswordValid($encoded, $raw, $salt)
    {
        if ($this->isPasswordTooLong($raw)) {
            return false;
        }

        // ...
}
```

Using Password Encoders

When the *getEncoder()*[22] method of the password encoder factory is called with the user object as its first argument, it will return an encoder of type *PasswordEncoderInterface*[23] which should be used to encode this user's password:

Listing 61-6

```php
// a Acme\Entity\LegacyUser instance
$user = ...;

// the password that was submitted, e.g. when registering
$plainPassword = ...;

$encoder = $encoderFactory->getEncoder($user);

// will return $weakEncoder (see above)
$encodedPassword = $encoder->encodePassword($plainPassword, $user->getSalt());

$user->setPassword($encodedPassword);

// ... save the user
```

Now, when you want to check if the submitted password (e.g. when trying to log in) is correct, you can use:

Listing 61-7

```php
// fetch the Acme\Entity\LegacyUser
$user = ...;

// the submitted password, e.g. from the login form
$plainPassword = ...;

$validPassword = $encoder->isPasswordValid(
```

22. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/EncoderFactory.html#getEncoder()
23. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
8       $user->getPassword(), // the encoded password
9       $plainPassword,       // the submitted password
10      $user->getSalt()
11  );
```

Chapter 62
Authorization

When any of the authentication providers (see *Authentication Providers*) has verified the still-unauthenticated token, an authenticated token will be returned. The authentication listener should set this token directly in the *`SecurityContextInterface`*[1] using its *`setToken()`*[2] method.

From then on, the user is authenticated, i.e. identified. Now, other parts of the application can use the token to decide whether or not the user may request a certain URI, or modify a certain object. This decision will be made by an instance of *`AccessDecisionManagerInterface`*[3].

An authorization decision will always be based on a few things:

- **The current token**

  For instance, the token's *`getRoles()`*[4] method may be used to retrieve the roles of the current user (e.g. `ROLE_SUPER_ADMIN`), or a decision may be based on the class of the token.

- **A set of attributes**

  Each attribute stands for a certain right the user should have, e.g. `ROLE_ADMIN` to make sure the user is an administrator.

- **An object (optional)**

  Any object for which access control needs to be checked, like an article or a comment object.

Access Decision Manager

Since deciding whether or not a user is authorized to perform a certain action can be a complicated process, the standard *`AccessDecisionManager`*[5] itself depends on multiple voters, and makes a final

1. http://api.symfony.com/2.3/Symfony/Component/Security/Core/SecurityContextInterface.html
2. http://api.symfony.com/2.3/Symfony/Component/Security/Core/SecurityContextInterface.html#setToken()
3. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/AccessDecisionManagerInterface.html
4. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#getRoles()
5. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/AccessDecisionManager.html

verdict based on all the votes (either positive, negative or neutral) it has received. It recognizes several strategies:

`affirmative` (default)
: grant access as soon as any voter returns an affirmative response;

`consensus`
: grant access if there are more voters granting access than there are denying;

`unanimous`
: only grant access if none of the voters has denied access;

Listing 62-1
```php
use Symfony\Component\Security\Core\Authorization\AccessDecisionManager;

// instances of Symfony\Component\Security\Core\Authorization\Voter\VoterInterface
$voters = array(...);

// one of "affirmative", "consensus", "unanimous"
$strategy = ...;

// whether or not to grant access when all voters abstain
$allowIfAllAbstainDecisions = ...;

// whether or not to grant access when there is no majority (applies only to the
"consensus" strategy)
$allowIfEqualGrantedDeniedDecisions = ...;

$accessDecisionManager = new AccessDecisionManager(
    $voters,
    $strategy,
    $allowIfAllAbstainDecisions,
    $allowIfEqualGrantedDeniedDecisions
);
```

You can change the default strategy in the configuration.

Voters

Voters are instances of *`VoterInterface`*[6], which means they have to implement a few methods which allows the decision manager to use them:

`supportsAttribute($attribute)`
: will be used to check if the voter knows how to handle the given attribute;

`supportsClass($class)`
: will be used to check if the voter is able to grant or deny access for an object of the given class;

`vote(TokenInterface $token, $object, array $attributes)`
: this method will do the actual voting and return a value equal to one of the class constants of *`VoterInterface`*[7], i.e. `VoterInterface::ACCESS_GRANTED`, `VoterInterface::ACCESS_DENIED` or `VoterInterface::ACCESS_ABSTAIN`;

The Security component contains some standard voters which cover many use cases:

6. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html
7. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html

AuthenticatedVoter

The *AuthenticatedVoter*[8] voter supports the attributes `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY` and grants access based on the current level of authentication, i.e. is the user fully authenticated, or only based on a "remember-me" cookie, or even authenticated anonymously?

Listing 62-2
```php
use Symfony\Component\Security\Core\Authentication\AuthenticationTrustResolver;

$anonymousClass = 'Symfony\Component\Security\Core\Authentication\Token\AnonymousToken';
$rememberMeClass = 'Symfony\Component\Security\Core\Authentication\Token\RememberMeToken';

$trustResolver = new AuthenticationTrustResolver($anonymousClass, $rememberMeClass);

$authenticatedVoter = new AuthenticatedVoter($trustResolver);

// instance of Symfony\Component\Security\Core\Authentication\Token\TokenInterface
$token = ...;

// any object
$object = ...;

$vote = $authenticatedVoter->vote($token, $object, array('IS_AUTHENTICATED_FULLY'));
```

RoleVoter

The *RoleVoter*[9] supports attributes starting with `ROLE_` and grants access to the user when the required `ROLE_*` attributes can all be found in the array of roles returned by the token's *getRoles()*[10] method:

Listing 62-3
```php
use Symfony\Component\Security\Core\Authorization\Voter\RoleVoter;

$roleVoter = new RoleVoter('ROLE_');

$roleVoter->vote($token, $object, array('ROLE_ADMIN'));
```

RoleHierarchyVoter

The *RoleHierarchyVoter*[11] extends *RoleVoter*[12] and provides some additional functionality: it knows how to handle a hierarchy of roles. For instance, a `ROLE_SUPER_ADMIN` role may have subroles `ROLE_ADMIN` and `ROLE_USER`, so that when a certain object requires the user to have the `ROLE_ADMIN` role, it grants access to users who in fact have the `ROLE_ADMIN` role, but also to users having the `ROLE_SUPER_ADMIN` role:

Listing 62-4
```php
use Symfony\Component\Security\Core\Authorization\Voter\RoleHierarchyVoter;
use Symfony\Component\Security\Core\Role\RoleHierarchy;

$hierarchy = array(
    'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_USER'),
);
```

8. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/AuthenticatedVoter.html
9. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html
10. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#getRoles()
11. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/RoleHierarchyVoter.html
12. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html

```
7
8  $roleHierarchy = new RoleHierarchy($hierarchy);
9
10 $roleHierarchyVoter = new RoleHierarchyVoter($roleHierarchy);
```

When you make your own voter, you may of course use its constructor to inject any dependencies it needs to come to a decision.

Roles

Roles are objects that give expression to a certain right the user has. The only requirement is that they implement *`RoleInterface`*[13], which means they should also have a *`getRole()`*[14] method that returns a string representation of the role itself. The default *`Role`*[15] simply returns its first constructor argument:

Listing 62-5
```
1  use Symfony\Component\Security\Core\Role\Role;
2
3  $role = new Role('ROLE_ADMIN');
4
5  // will echo 'ROLE_ADMIN'
6  echo $role->getRole();
```

Most authentication tokens extend from *`AbstractToken`*[16], which means that the roles given to its constructor will be automatically converted from strings to these simple `Role` objects.

Using the Decision Manager

The Access Listener

The access decision manager can be used at any point in a request to decide whether or not the current user is entitled to access a given resource. One optional, but useful, method for restricting access based on a URL pattern is the *`AccessListener`*[17], which is one of the firewall listeners (see *Firewall Listeners*) that is triggered for each request matching the firewall map (see *A Firewall for HTTP Requests*).

It uses an access map (which should be an instance of *`AccessMapInterface`*[18]) which contains request matchers and a corresponding set of attributes that are required for the current user to get access to the application:

Listing 62-6
```
1  use Symfony\Component\Security\Http\AccessMap;
2  use Symfony\Component\HttpFoundation\RequestMatcher;
3  use Symfony\Component\Security\Http\Firewall\AccessListener;
```

13. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Role/RoleInterface.html
14. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Role/Role/RoleInterface.html#getRole()
15. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Role/Role.html
16. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Authentication/Token/AbstractToken.html
17. http://api.symfony.com/2.3/Symfony/Component/Security/Http/Firewall/AccessListener.html
18. http://api.symfony.com/2.3/Symfony/Component/Security/Http/AccessMapInterface.html

```
4
5  $accessMap = new AccessMap();
6  $requestMatcher = new RequestMatcher('^/admin');
7  $accessMap->add($requestMatcher, array('ROLE_ADMIN'));
8
9  $accessListener = new AccessListener(
10     $securityContext,
11     $accessDecisionManager,
12     $accessMap,
13     $authenticationManager
14 );
```

Security Context

The access decision manager is also available to other parts of the application via the *isGranted()*[19] method of the *SecurityContext*[20]. A call to this method will directly delegate the question to the access decision manager:

Listing 62-7

```
1  use Symfony\Component\Security\SecurityContext;
2  use Symfony\Component\Security\Core\Exception\AccessDeniedException;
3
4  $securityContext = new SecurityContext(
5      $authenticationManager,
6      $accessDecisionManager
7  );
8
9  if (!$securityContext->isGranted('ROLE_ADMIN')) {
10     throw new AccessDeniedException();
11 }
```

19. http://api.symfony.com/2.3/Symfony/Component/Security/Core/SecurityContext.html#isGranted()
20. http://api.symfony.com/2.3/Symfony/Component/Security/Core/SecurityContext.html

Chapter 63

Securely Comparing Strings and Generating Random Numbers

The Symfony Security component comes with a collection of nice utilities related to security. These utilities are used by Symfony, but you should also use them if you want to solve the problem they address.

Comparing Strings

The time it takes to compare two strings depends on their differences. This can be used by an attacker when the two strings represent a password for instance; it is known as a *Timing attack*[1].

Internally, when comparing two passwords, Symfony uses a constant-time algorithm; you can use the same strategy in your own code thanks to the *StringUtils*[2] class:

Listing 63-1
```php
use Symfony\Component\Security\Core\Util\StringUtils;

// is some known string (e.g. password) equal to some user input?
$bool = StringUtils::equals($knownString, $userInput);
```

To avoid timing attacks, the known string must be the first argument and the user-entered string the second.

Generating a Secure random Number

Whenever you need to generate a secure random number, you are highly encouraged to use the Symfony *SecureRandom*[3] class:

1. http://en.wikipedia.org/wiki/Timing_attack
2. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Util/StringUtils.html
3. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Util/SecureRandom.html

Listing 63-2
```php
use Symfony\Component\Security\Core\Util\SecureRandom;

$generator = new SecureRandom();
$random = $generator->nextBytes(10);
```

The *nextBytes()*[4] method returns a random string composed of the number of characters passed as an argument (10 in the above example).

The SecureRandom class works better when OpenSSL is installed. But when it's not available, it falls back to an internal algorithm, which needs a seed file to work correctly. Just pass a file name to enable it:

Listing 63-3
```php
use Symfony\Component\Security\Core\Util\SecureRandom;

$generator = new SecureRandom('/some/path/to/store/the/seed.txt');
$random = $generator->nextBytes(10);
```

If you're using the Symfony Framework, you can access a secure random instance directly from the container: its name is `security.secure_random`.

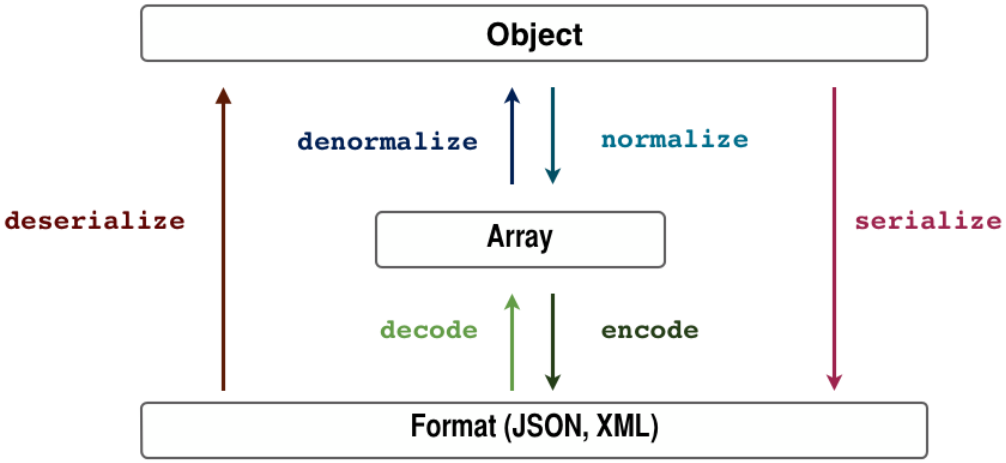
4. http://api.symfony.com/2.3/Symfony/Component/Security/Core/Util/SecureRandom.html#nextBytes()

Chapter 64

The Serializer Component

> The Serializer component is meant to be used to turn objects into a specific format (XML, JSON, YAML, ...) and the other way around.

In order to do so, the Serializer component follows the following simple schema.



As you can see in the picture above, an array is used as a man in the middle. This way, Encoders will only deal with turning specific **formats** into **arrays** and vice versa. The same way, Normalizers will deal with turning specific **objects** into **arrays** and vice versa.

Serialization is a complicated topic, and while this component may not work in all cases, it can be a useful tool while developing tools to serialize and deserialize your objects.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/serializer` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Serializer*[2]).

Usage

Using the Serializer component is really simple. You just need to set up the *`Serializer`*[3] specifying which Encoders and Normalizer are going to be available:

Listing 64-1
```php
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;

$encoders = array(new XmlEncoder(), new JsonEncoder());
$normalizers = array(new GetSetMethodNormalizer());

$serializer = new Serializer($normalizers, $encoders);
```

Serializing an Object

For the sake of this example, assume the following class already exists in your project:

Listing 64-2
```php
namespace Acme;

class Person
{
    private $age;
    private $name;

    // Getters
    public function getName()
    {
        return $this->name;
    }

    public function getAge()
    {
        return $this->age;
    }

    // Setters
    public function setName($name)
    {
        $this->name = $name;
    }

    public function setAge($age)
    {
        $this->age = $age;
    }
}
```

1. https://packagist.org/packages/symfony/serializer
2. https://github.com/symfony/Serializer
3. http://api.symfony.com/2.3/Symfony/Component/Serializer/Serializer.html

Now, if you want to serialize this object into JSON, you only need to use the Serializer service created before:

Listing 64-3
```php
$person = new Acme\Person();
$person->setName('foo');
$person->setAge(99);

$jsonContent = $serializer->serialize($person, 'json');

// $jsonContent contains {"name":"foo","age":99}

echo $jsonContent; // or return it in a Response
```

The first parameter of the *serialize()*[4] is the object to be serialized and the second is used to choose the proper encoder, in this case *JsonEncoder*[5].

Ignoring Attributes when Serializing

New in version 2.3: The *GetSetMethodNormalizer::setIgnoredAttributes*[6] method was introduced in Symfony 2.3.

As an option, there's a way to ignore attributes from the origin object when serializing. To remove those attributes use the *setIgnoredAttributes()*[7] method on the normalizer definition:

Listing 64-4
```php
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;

$normalizer = new GetSetMethodNormalizer();
$normalizer->setIgnoredAttributes(array('age'));
$encoder = new JsonEncoder();

$serializer = new Serializer(array($normalizer), array($encoder));
$serializer->serialize($person, 'json'); // Output: {"name":"foo"}
```

Deserializing an Object

You'll now learn how to do the exact opposite. This time, the information of the `Person` class would be encoded in XML format:

Listing 64-5
```php
$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
</person>
EOF;

$person = $serializer->deserialize($data, 'Acme\Person', 'xml');
```

In this case, *deserialize()*[8] needs three parameters:

4. http://api.symfony.com/2.3/Symfony/Component/Serializer/Serializer.html#serialize()
5. http://api.symfony.com/2.3/Symfony/Component/Serializer/Encoder/JsonEncoder.html
6. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html#setIgnoredAttributes()
7. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html#setIgnoredAttributes()
8. http://api.symfony.com/2.3/Symfony/Component/Serializer/Serializer.html#deserialize()

1. The information to be decoded
2. The name of the class this information will be decoded to
3. The encoder used to convert that information into an array

Using Camelized Method Names for Underscored Attributes

New in version 2.3: The *`GetSetMethodNormalizer::setCamelizedAttributes`*[9] method was introduced in Symfony 2.3.

Sometimes property names from the serialized content are underscored (e.g. `first_name`). Normally, these attributes will use get/set methods like `getFirst_name`, when `getFirstName` method is what you really want. To change that behavior use the *`setCamelizedAttributes()`*[10] method on the normalizer definition:

Listing 64-6
```php
$encoder = new JsonEncoder();
$normalizer = new GetSetMethodNormalizer();
$normalizer->setCamelizedAttributes(array('first_name'));

$serializer = new Serializer(array($normalizer), array($encoder));

$json = <<<EOT
{
    "name":       "foo",
    "age":        "19",
    "first_name": "bar"
}
EOT;

$person = $serializer->deserialize($json, 'Acme\Person', 'json');
```

As a final result, the deserializer uses the `first_name` attribute as if it were `firstName` and uses the `getFirstName` and `setFirstName` methods.

Using Callbacks to Serialize Properties with Object Instances

When serializing, you can set a callback to format a specific object property:

Listing 64-7
```php
use Acme\Person;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;
use Symfony\Component\Serializer\Serializer;

$encoder = new JsonEncoder();
$normalizer = new GetSetMethodNormalizer();

$callback = function ($dateTime) {
    return $dateTime instanceof \DateTime
        ? $dateTime->format(\DateTime::ISO8601)
        : '';
};

$normalizer->setCallbacks(array('createdAt' => $callback));
```

9. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html#setCamelizedAttributes()
10. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html#setCamelizedAttributes()

```php

$serializer = new Serializer(array($normalizer), array($encoder));

$person = new Person();
$person->setName('cordoval');
$person->setAge(34);
$person->setCreatedAt(new \DateTime('now'));

$serializer->serialize($person, 'json');
// Output: {"name":"cordoval", "age": 34, "createdAt": "2014-03-22T09:43:12-0500"}
```

JMSSerializer

A popular third-party library, *JMS serializer*[11], provides a more sophisticated albeit more complex solution. This library includes the ability to configure how your objects should be serialized/deserialized via annotations (as well as YAML, XML and PHP), integration with the Doctrine ORM, and handling of other complex cases (e.g. circular references).

11. https://github.com/schmittjoh/serializer

Chapter 65
The Stopwatch Component

> The Stopwatch component provides a way to profile code.

New in version 2.2: The Stopwatch component was introduced in Symfony 2.2. Previously, the `Stopwatch` class was located in the HttpKernel component (and was introduced in Symfony 2.1).

Installation

You can install the component in two different ways:

- *Install it via Composer* (`symfony/stopwatch` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Stopwatch*[2]).

Usage

The Stopwatch component provides an easy and consistent way to measure execution time of certain parts of code so that you don't constantly have to parse microtime by yourself. Instead, use the simple *`Stopwatch`*[3] class:

Listing 65-1
```php
use Symfony\Component\Stopwatch\Stopwatch;

$stopwatch = new Stopwatch();
// Start event named 'eventName'
$stopwatch->start('eventName');
// ... some code goes here
$event = $stopwatch->stop('eventName');
```

1. https://packagist.org/packages/symfony/stopwatch
2. https://github.com/symfony/Stopwatch
3. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html

The *StopwatchEvent*[4] object can be retrieved from the *start()*[5], *stop()*[6] and *lap()*[7] methods.

You can also provide a category name to an event:

Listing 65-2
```
$stopwatch->start('eventName', 'categoryName');
```

You can consider categories as a way of tagging events. For example, the Symfony Profiler tool uses categories to nicely color-code different events.

Periods

As you know from the real world, all stopwatches come with two buttons: one to start and stop the stopwatch, and another to measure the lap time. This is exactly what the *lap()*[8] method does:

Listing 65-3
```
$stopwatch = new Stopwatch();
// Start event named 'foo'
$stopwatch->start('foo');
// ... some code goes here
$stopwatch->lap('foo');
// ... some code goes here
$stopwatch->lap('foo');
// ... some other code goes here
$event = $stopwatch->stop('foo');
```

Lap information is stored as "periods" within the event. To get lap information call:

Listing 65-4
```
$event->getPeriods();
```

In addition to periods, you can get other useful information from the event object. For example:

Listing 65-5
```
$event->getCategory();    // Returns the category the event was started in
$event->getOrigin();      // Returns the event start time in milliseconds
$event->ensureStopped(); // Stops all periods not already stopped
$event->getStartTime();   // Returns the start time of the very first period
$event->getEndTime();     // Returns the end time of the very last period
$event->getDuration();    // Returns the event duration, including all periods
$event->getMemory();      // Returns the max memory usage of all periods
```

Sections

Sections are a way to logically split the timeline into groups. You can see how Symfony uses sections to nicely visualize the framework lifecycle in the Symfony Profiler tool. Here is a basic usage example using sections:

Listing 65-6
```
$stopwatch = new Stopwatch();

$stopwatch->openSection();
```

4. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/StopwatchEvent.html
5. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html#start()
6. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html#stop()
7. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html#lap()
8. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html#lap()

```
4 $stopwatch->start('parsing_config_file', 'filesystem_operations');
5 $stopwatch->stopSection('routing');
6
7 $events = $stopwatch->getSectionEvents('routing');
```

You can reopen a closed section by calling the *openSection()*[9] method and specifying the id of the section to be reopened:

Listing 65-7
```
1 $stopwatch->openSection('routing');
2 $stopwatch->start('building_config_tree');
3 $stopwatch->stopSection('routing');
```

9. http://api.symfony.com/2.3/Symfony/Component/Stopwatch/Stopwatch.html#openSection()

Chapter 66
The Templating Component

> The Templating component provides all the tools needed to build any kind of template system.
>
> It provides an infrastructure to load template files and optionally monitor them for changes. It also provides a concrete template engine implementation using PHP with additional tools for escaping and separating templates into blocks and layouts.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/templating` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Templating*[2]).

Usage

The *`PhpEngine`*[3] class is the entry point of the component. It needs a template name parser (*`TemplateNameParserInterface`*[4]) to convert a template name to a template reference (*`TemplateReferenceInterface`*[5]). It also needs a template loader (*`LoaderInterface`*[6]) which uses the template reference to actually find and load the template:

Listing 66-1
```
use Symfony\Component\Templating\PhpEngine;
use Symfony\Component\Templating\TemplateNameParser;
use Symfony\Component\Templating\Loader\FilesystemLoader;
```

1. https://packagist.org/packages/symfony/templating
2. https://github.com/symfony/Templating
3. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html
4. http://api.symfony.com/2.3/Symfony/Component/Templating/TemplateNameParserInterface.html
5. http://api.symfony.com/2.3/Symfony/Component/Templating/TemplateReferenceInterface.html
6. http://api.symfony.com/2.3/Symfony/Component/Templating/Loader/LoaderInterface.html

```
4
5  $loader = new FilesystemLoader(__DIR__.'/views/%name%');
6
7  $templating = new PhpEngine(new TemplateNameParser(), $loader);
8
9  echo $templating->render('hello.php', array('firstname' => 'Fabien'));
```

Listing 66-2
```
1  <!-- views/hello.php -->
2  Hello, <?php echo $firstname ?>!
```

The *render()*[7] method parses the `views/hello.php` file and returns the output text. The second argument of `render` is an array of variables to use in the template. In this example, the result will be `Hello, Fabien!`.

Templates will be cached in the memory of the engine. This means that if you render the same template multiple times in the same request, the template will only be loaded once from the file system.

The `$view` Variable

In all templates parsed by the `PhpEngine`, you get access to a mysterious variable called `$view`. That variable holds the current `PhpEngine` instance. That means you get access to a bunch of methods that make your life easier.

Including Templates

The best way to share a snippet of template code is to create a template that can then be included by other templates. As the `$view` variable is an instance of `PhpEngine`, you can use the `render` method (which was used to render the template originally) inside the template to render another template:

Listing 66-3
```
1  <?php $names = array('Fabien', ...) ?>
2  <?php foreach ($names as $name) : ?>
3      <?php echo $view->render('hello.php', array('firstname' => $name)) ?>
4  <?php endforeach ?>
```

Global Variables

Sometimes, you need to set a variable which is available in all templates rendered by an engine (like the `$app` variable when using the Symfony framework). These variables can be set by using the *addGlobal()*[8] method and they can be accessed in the template as normal variables:

Listing 66-4
```
1  $templating->addGlobal('ga_tracking', 'UA-xxxxx-x');
```

In a template:

7. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#render()
8. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#addGlobal()

Listing 66-5
```
<p>The google tracking code is: <?php echo $ga_tracking ?></p>
```

The global variables cannot be called `this` or `view`, since they are already used by the PHP engine.

The global variables can be overridden by a local variable in the template with the same name.

Output Escaping

When you render variables, you should probably escape them so that HTML or JavaScript code isn't written out to your page. This will prevent things like XSS attacks. To do this, use the *`escape()`*[9] method:

Listing 66-6
```
<?php echo $view->escape($firstname) ?>
```

By default, the `escape()` method assumes that the variable is outputted within an HTML context. The second argument lets you change the context. For example, to output something inside JavaScript, use the `js` context:

Listing 66-7
```
<?php echo $view->escape($var, 'js') ?>
```

The component comes with an HTML and JS escaper. You can register your own escaper using the *`setEscaper()`*[10] method:

Listing 66-8
```
$templating->setEscaper('css', function ($value) {
    // ... all CSS escaping

    return $escapedValue;
});
```

Helpers

The Templating component can be easily extended via helpers. Helpers are PHP objects that provide features useful in a template context. The component has 2 built-in helpers:

- *Assets Helper*
- *Slots Helper*

Before you can use these helpers, you need to register them using *`set()`*[11]:

Listing 66-9

9. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#escape()
10. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#setEscaper()
11. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#set()

```php
use Symfony\Component\Templating\Helper\AssetsHelper;
// ...

$templating->set(new AssetsHelper());
```

Custom Helpers

You can create your own helpers by creating a class which implements *HelperInterface*[12]. However, most of the time you'll extend *Helper*[13].

The `Helper` has one required method: *getName()*[14]. This is the name that is used to get the helper from the `$view` object.

Creating a Custom Engine

Besides providing a PHP templating engine, you can also create your own engine using the Templating component. To do that, create a new class which implements the *EngineInterface*[15]. This requires 3 method:

- *render($name, array $parameters = array())*[16] - Renders a template
- *exists($name)*[17] - Checks if the template exists
- *supports($name)*[18] - Checks if the given template can be handled by this engine.

Using Multiple Engines

It is possible to use multiple engines at the same time using the *DelegatingEngine*[19] class. This class takes a list of engines and acts just like a normal templating engine. The only difference is that it delegates the calls to one of the other engines. To choose which one to use for the template, the *EngineInterface::supports()*[20] method is used.

Listing 66-10
```php
use Acme\Templating\CustomEngine;
use Symfony\Component\Templating\PhpEngine;
use Symfony\Component\Templating\DelegatingEngine;

$templating = new DelegatingEngine(array(
    new PhpEngine(...),
    new CustomEngine(...),
));
```

12. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/HelperInterface.html
13. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/Helper.html
14. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/HelperInterface.html#getName()
15. http://api.symfony.com/2.3/Symfony/Component/Templating/EngineInterface.html
16. http://api.symfony.com/2.3/Symfony/Component/Templating/EngineInterface.html#render()
17. http://api.symfony.com/2.3/Symfony/Component/Templating/EngineInterface.html#exists()
18. http://api.symfony.com/2.3/Symfony/Component/Templating/EngineInterface.html#supports()
19. http://api.symfony.com/2.3/Symfony/Component/Templating/DelegatingEngine.html
20. http://api.symfony.com/2.3/Symfony/Component/Templating/EngineInterface.html#supports()

Chapter 67
Slots Helper

More often than not, templates in a project share common elements, like the well-known header and footer. Using this helper, the static HTML code can be placed in a layout file along with "slots", which represent the dynamic parts that will change on a page-by-page basis. These slots are then filled in by different children template. In other words, the layout file decorates the child template.

Displaying Slots

The slots are accessible by using the slots helper (`$view['slots']`). Use *`output()`*[1] to display the content of the slot on that place:

Listing 67-1
```php
<!-- views/layout.php -->
<!doctype html>
<html>
    <head>
        <title>
            <?php $view['slots']->output('title', 'Default title') ?>
        </title>
    </head>
    <body>
        <?php $view['slots']->output('_content') ?>
    </body>
</html>
```

The first argument of the method is the name of the slot. The method has an optional second argument, which is the default value to use if the slot is not available.

The `_content` slot is a special slot set by the `PhpEngine`. It contains the content of the subtemplate.

1. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/SlotsHelper.html#output()

If you're using the standalone component, make sure you registered the *SlotsHelper*[2]:

Listing 67-2
```php
use Symfony\Component\Templating\Helper\SlotsHelper;

// ...
$templateEngine->set(new SlotsHelper());
```

Extending Templates

The *extend()*[3] method is called in the sub-template to set its parent template. Then *$view['slots']->set()*[4] can be used to set the content of a slot. All content which is not explicitly set in a slot is in the `_content` slot.

Listing 67-3
```php
<!-- views/page.php -->
<?php $view->extend('layout.php') ?>

<?php $view['slots']->set('title', $page->title) ?>

<h1>
    <?php echo $page->title ?>
</h1>
<p>
    <?php echo $page->body ?>
</p>
```

Multiple levels of inheritance is possible: a layout can extend another layout.

For large slots, there is also an extended syntax:

Listing 67-4
```php
<?php $view['slots']->start('title') ?>
    Some large amount of HTML
<?php $view['slots']->stop() ?>
```

2. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/SlotsHelper.html
3. http://api.symfony.com/2.3/Symfony/Component/Templating/PhpEngine.html#extend()
4. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/SlotsHelper.html#set()

Chapter 68
Assets Helper

The assets helper's main purpose is to make your application more portable by generating asset paths:

Listing 68-1
```
<link href="<?php echo $view['assets']->getUrl('css/style.css') ?>" rel="stylesheet">

<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>">
```

The assets helper can then be configured to render paths to a CDN or modify the paths in case your assets live in a sub-directory of your host (e.g. `http://example.com/app`).

Configure Paths

By default, the assets helper will prefix all paths with a slash. You can configure this by passing a base assets path as the first argument of the constructor:

Listing 68-2
```
use Symfony\Component\Templating\Helper\AssetsHelper;

// ...
$templateEngine->set(new AssetsHelper('/foo/bar'));
```

Now, if you use the helper, everything will be prefixed with `/foo/bar`:

Listing 68-3
```
<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>">
<!-- renders as:
<img src="/foo/bar/images/logo.png">
-->
```

Absolute Urls

You can also specify a URL to use in the second parameter of the constructor:

Listing 68-4

```php
// ...
$templateEngine->set(new AssetsHelper(null, 'http://cdn.example.com/'));
```

Now URLs are rendered like `http://cdn.example.com/images/logo.png`.

Versioning

To avoid using the cached resource after updating the old resource, you can use versions which you bump every time you release a new project. The version can be specified in the third argument:

Listing 68-5
```php
// ...
$templateEngine->set(new AssetsHelper(null, null, '328rad75'));
```

Now, every URL is suffixed with `?328rad75`. If you want to have a different format, you can specify the new format in fourth argument. It's a string that is used in *sprintf*[1]. The first argument is the path and the second is the version. For instance, `%s?v=%s` will be rendered as `/images/logo.png?v=328rad75`.

Multiple Packages

Asset path generation is handled internally by packages. The component provides 2 packages by default:

- *PathPackage*[2]
- *UrlPackage*[3]

You can also use multiple packages:

Listing 68-6
```php
use Symfony\Component\Templating\Asset\PathPackage;

// ...
$templateEngine->set(new AssetsHelper());

$templateEngine->get('assets')->addPackage('images', new PathPackage('/images/'));
$templateEngine->get('assets')->addPackage('scripts', new PathPackage('/scripts/'));
```

This will setup the assets helper with 3 packages: the default package which defaults to `/` (set by the constructor), the images package which prefixes it with `/images/` and the scripts package which prefixes it with `/scripts/`.

If you want to set another default package, you can use *setDefaultPackage()*[4].

You can specify which package you want to use in the second argument of *getUrl()*[5]:

Listing 68-7
```php
<img src="<?php echo $view['assets']->getUrl('foo.png', 'images') ?>">
<!-- renders as:
<img src="/images/foo.png">
-->
```

1. http://php.net/manual/en/function.sprintf.php
2. http://api.symfony.com/2.3/Symfony/Component/Templating/Asset/PathPackage.html
3. http://api.symfony.com/2.3/Symfony/Component/Templating/Asset/UrlPackage.html
4. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/AssetsHelper.html#setDefaultPackage()
5. http://api.symfony.com/2.3/Symfony/Component/Templating/Helper/AssetsHelper.html#getUrl()

Custom Packages

You can create your own package by extending *Package*[6].

6. http://api.symfony.com/2.3/Symfony/Component/Templating/Asset/Package.html

Chapter 69
The Translation Component

> The Translation component provides tools to internationalize your application.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/translation` on *Packagist*[1]);
- Use the official Git repository (*https://github.com/symfony/Translation*[2]).

Constructing the Translator

The main access point of the Translation component is *`Translator`*[3]. Before you can use it, you need to configure it and load the messages to translate (called *message catalogs*).

Configuration

The constructor of the `Translator` class needs one argument: The locale.

Listing 69-1
```php
use Symfony\Component\Translation\Translator;
use Symfony\Component\Translation\MessageSelector;

$translator = new Translator('fr_FR', new MessageSelector());
```

1. https://packagist.org/packages/symfony/translation
2. https://github.com/symfony/Translation
3. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html

The locale set here is the default locale to use. You can override this locale when translating strings.

The term *locale* refers roughly to the user's language and country. It can be any string that your application uses to manage translations and other format differences (e.g. currency format). The *ISO 639-1*[4] *language* code, an underscore (_), then the *ISO 3166-1 alpha-2*[5] *country* code (e.g. `fr_FR` for French/France) is recommended.

Loading Message Catalogs

The messages are stored in message catalogs inside the `Translator` class. A message catalog is like a dictionary of translations for a specific locale.

The Translation component uses Loader classes to load catalogs. You can load multiple resources for the same locale, which will then be combined into one catalog.

The component comes with some default loaders:

- *`ArrayLoader`*[6] - to load catalogs from PHP arrays.
- *`CsvFileLoader`*[7] - to load catalogs from CSV files.
- *`IcuDatFileLoader`*[8] - to load catalogs from resource bundles.
- *`IcuResFileLoader`*[9] - to load catalogs from resource bundles.
- *`IniFileLoader`*[10] - to load catalogs from ini files.
- *`MoFileLoader`*[11] - to load catalogs from gettext files.
- *`PhpFileLoader`*[12] - to load catalogs from PHP files.
- *`PoFileLoader`*[13] - to load catalogs from gettext files.
- *`QtFileLoader`*[14] - to load catalogs from QT XML files.
- *`XliffFileLoader`*[15] - to load catalogs from Xliff files.
- *`YamlFileLoader`*[16] - to load catalogs from Yaml files (requires the *Yaml component*).

New in version 2.1: The `IcuDatFileLoader`, `IcuResFileLoader`, `IniFileLoader`, `MoFileLoader`, `PoFileLoader` and `QtFileLoader` were introduced in Symfony 2.1.

All file loaders require the *Config component*.

You can also *create your own Loader*, in case the format is not already supported by one of the default loaders.

At first, you should add one or more loaders to the `Translator`:

Listing 69-2
```
// ...
$translator->addLoader('array', new ArrayLoader());
```

4. http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
5. http://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
6. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/ArrayLoader.html
7. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/CsvFileLoader.html
8. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/IcuDatFileLoader.html
9. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/IcuResFileLoader.html
10. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/IniFileLoader.html
11. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/MoFileLoader.html
12. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/PhpFileLoader.html
13. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/PoFileLoader.html
14. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/QtFileLoader.html
15. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/XliffFileLoader.html
16. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/YamlFileLoader.html

The first argument is the name to which you can refer the loader in the translator and the second argument is an instance of the loader itself. After this, you can add your resources using the correct loader.

Loading Messages with the `ArrayLoader`

Loading messages can be done by calling *`addResource()`*[17]. The first argument is the loader name (this was the first argument of the `addLoader` method), the second is the resource and the third argument is the locale:

Listing 69-3
```
// ...
$translator->addResource('array', array(
    'Hello World!' => 'Bonjour',
), 'fr_FR');
```

Loading Messages with the File Loaders

If you use one of the file loaders, you should also use the `addResource` method. The only difference is that you should put the file name to the resource file as the second argument, instead of an array:

Listing 69-4
```
// ...
$translator->addLoader('yaml', new YamlFileLoader());
$translator->addResource('yaml', 'path/to/messages.fr.yml', 'fr_FR');
```

The Translation Process

To actually translate the message, the Translator uses a simple process:

- A catalog of translated messages is loaded from translation resources defined for the `locale` (e.g. `fr_FR`). Messages from the *Fallback Locales* are also loaded and added to the catalog, if they don't already exist. The end result is a large "dictionary" of translations;
- If the message is located in the catalog, the translation is returned. If not, the translator returns the original message.

You start this process by calling *`trans()`*[18] or *`transChoice()`*[19]. Then, the Translator looks for the exact string inside the appropriate message catalog and returns it (if it exists).

Fallback Locales

If the message is not located in the catalog of the specific locale, the translator will look into the catalog of one or more fallback locales. For example, assume you're trying to translate into the `fr_FR` locale:

1. First, the translator looks for the translation in the `fr_FR` locale;
2. If it wasn't found, the translator looks for the translation in the `fr` locale;
3. If the translation still isn't found, the translator uses the one or more fallback locales set explicitly on the translator.

For (3), the fallback locales can be set by calling *`setFallbackLocale()`*[20]:

Listing 69-5

17. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html#addResource()
18. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html#trans()
19. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html#transChoice()
20. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html#setFallbackLocale()

```
// ...
$translator->setFallbackLocale(array('en'));
```

Using Message Domains

As you've seen, message files are organized into the different locales that they translate. The message files can also be organized further into "domains".

The domain is specified in the fourth argument of the `addResource()` method. The default domain is `messages`. For example, suppose that, for organization, translations were split into three different domains: `messages`, `admin` and `navigation`. The French translation would be loaded like this:

Listing 69-6

```
// ...
$translator->addLoader('xlf', new XliffFileLoader());

$translator->addResource('xlf', 'messages.fr.xlf', 'fr_FR');
$translator->addResource('xlf', 'admin.fr.xlf', 'fr_FR', 'admin');
$translator->addResource(
    'xlf',
    'navigation.fr.xlf',
    'fr_FR',
    'navigation'
);
```

When translating strings that are not in the default domain (`messages`), you must specify the domain as the third argument of `trans()`:

Listing 69-7

```
$translator->trans('Symfony is great', array(), 'admin');
```

Symfony will now look for the message in the `admin` domain of the specified locale.

Usage

Read how to use the Translation component in *Using the Translator*.

Chapter 70
Using the Translator

Imagine you want to translate the string *"Symfony is great"* into French:

Listing 70-1
```php
use Symfony\Component\Translation\Translator;
use Symfony\Component\Translation\Loader\ArrayLoader;

$translator = new Translator('fr_FR');
$translator->addLoader('array', new ArrayLoader());
$translator->addResource('array', array(
    'Symfony is great!' => 'J\'aime Symfony!',
), 'fr_FR');

echo $translator->trans('Symfony is great!');
```

In this example, the message *"Symfony is great!"* will be translated into the locale set in the constructor (`fr_FR`) if the message exists in one of the message catalogs.

Message Placeholders

Sometimes, a message containing a variable needs to be translated:

Listing 70-2
```php
// ...
$translated = $translator->trans('Hello '.$name);

echo $translated;
```

However, creating a translation for this string is impossible since the translator will try to look up the exact message, including the variable portions (e.g. *"Hello Ryan"* or *"Hello Fabien"*). Instead of writing a translation for every possible iteration of the `$name` variable, you can replace the variable with a "placeholder":

Listing 70-3
```php
// ...
$translated = $translator->trans(
```

```
    'Hello %name%',
    array('%name%' => $name)
);

echo $translated;
```

Symfony will now look for a translation of the raw message (`Hello %name%`) and *then* replace the placeholders with their values. Creating a translation is done just as before:

Listing 70-4
```xml
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Bonjour %name%</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

The placeholders can take on any form as the full message is reconstructed using the PHP *`strtr function`*[1]. But the `%...%` form is recommended, to avoid problems when using Twig.

As you've seen, creating a translation is a two-step process:

1. Abstract the message that needs to be translated by processing it through the `Translator`.
2. Create a translation for the message in each locale that you choose to support.

The second step is done by creating message catalogs that define the translations for any number of different locales.

Creating Translations

The act of creating translation files is an important part of "localization" (often abbreviated *L10n*[2]). Translation files consist of a series of id-translation pairs for the given domain and locale. The source is the identifier for the individual translation, and can be the message in the main locale (e.g. *"Symfony is great"*) of your application or a unique identifier (e.g. `symfony.great` - see the sidebar below).

Translation files can be created in several different formats, XLIFF being the recommended format. These files are parsed by one of the loader classes.

Listing 70-5
```xml
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony is great</source>
                <target>J'aime Symfony</target>
            </trans-unit>
            <trans-unit id="2">
```

1. http://php.net/manual/en/function.strtr.php
2. http://en.wikipedia.org/wiki/Internationalization_and_localization

```
                <source>symfony.great</source>
                <target>J'aime Symfony</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Using Real or Keyword Messages

This example illustrates the two different philosophies when creating messages to be translated:

Listing 70-6
```
$translator->trans('Symfony is great');

$translator->trans('symfony.great');
```

In the first method, messages are written in the language of the default locale (English in this case). That message is then used as the "id" when creating translations.

In the second method, messages are actually "keywords" that convey the idea of the message. The keyword message is then used as the "id" for any translations. In this case, translations must be made for the default locale (i.e. to translate `symfony.great` to `Symfony is great`).

The second method is handy because the message key won't need to be changed in every translation file if you decide that the message should actually read "Symfony is really great" in the default locale.

The choice of which method to use is entirely up to you, but the "keyword" format is often recommended.

Additionally, the `php` and `yaml` file formats support nested ids to avoid repeating yourself if you use keywords instead of real text for your ids:

Listing 70-7
```
symfony:
    is:
        great: Symfony is great
        amazing: Symfony is amazing
    has:
        bundles: Symfony has bundles
user:
    login: Login
```

The multiple levels are flattened into single id/translation pairs by adding a dot (`.`) between every level, therefore the above examples are equivalent to the following:

Listing 70-8
```
symfony.is.great: Symfony is great
symfony.is.amazing: Symfony is amazing
symfony.has.bundles: Symfony has bundles
user.login: Login
```

Pluralization

Message pluralization is a tough topic as the rules can be quite complex. For instance, here is the mathematical representation of the Russian pluralization rules:

Listing 70-9

```
((($number % 10 == 1) && ($number % 100 != 11))
    ? 0
    : ((($number % 10 >= 2)
        && ($number % 10 <= 4)
        && (($number % 100 < 10)
        || ($number % 100 >= 20)))
            ? 1
            : 2
);
```

As you can see, in Russian, you can have three different plural forms, each given an index of 0, 1 or 2. For each form, the plural is different, and so the translation is also different.

When a translation has different forms due to pluralization, you can provide all the forms as a string separated by a pipe (`|`):

Listing 70-10
```
'There is one apple|There are %count% apples'
```

To translate pluralized messages, use the *`transChoice()`*[3] method:

Listing 70-11
```
$translator->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

The second argument (`10` in this example) is the *number* of objects being described and is used to determine which translation to use and also to populate the `%count%` placeholder.

Based on the given number, the translator chooses the right plural form. In English, most words have a singular form when there is exactly one object and a plural form for all other numbers (0, 2, 3...). So, if `count` is `1`, the translator will use the first string (`There is one apple`) as the translation. Otherwise it will use `There are %count% apples`.

Here is the French translation:

Listing 70-12
```
'Il y a %count% pomme|Il y a %count% pommes'
```

Even if the string looks similar (it is made of two sub-strings separated by a pipe), the French rules are different: the first form (no plural) is used when `count` is `0` or `1`. So, the translator will automatically use the first string (`Il y a %count% pomme`) when `count` is `0` or `1`.

Each locale has its own set of rules, with some having as many as six different plural forms with complex rules behind which numbers map to which plural form. The rules are quite simple for English and French, but for Russian, you'd may want a hint to know which rule matches which string. To help translators, you can optionally "tag" each string:

Listing 70-13
```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

The tags are really only hints for translators and don't affect the logic used to determine which plural form to use. The tags can be any descriptive string that ends with a colon (`:`). The tags also do not need to be the same in the original message as in the translated one.

3. http://api.symfony.com/2.3/Symfony/Component/Translation/Translator.html#transChoice()

As tags are optional, the translator doesn't use them (the translator will only get a string based on its position in the string).

Explicit Interval Pluralization

The easiest way to pluralize a message is to let the Translator use internal logic to choose which string to use based on a given number. Sometimes, you'll need more control or want a different translation for specific cases (for `0`, or when the count is negative, for example). For such cases, you can use explicit math intervals:

Listing 70-14
```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf]
There are many apples'
```

The intervals follow the *ISO 31-11*[4] notation. The above string specifies four different intervals: exactly `0`, exactly `1`, `2-19`, and `20` and higher.

You can also mix explicit math rules and standard rules. In this case, if the count is not matched by a specific interval, the standard rules take effect after removing the explicit rules:

Listing 70-15
```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are
%count% apples'
```

For example, for `1` apple, the standard rule `There is one apple` will be used. For `2-19` apples, the second standard rule `There are %count% apples` will be selected.

An *`Interval`*[5] can represent a finite set of numbers:

Listing 70-16
```
{1,2,3,4}
```

Or numbers between two other numbers:

Listing 70-17
```
[1, +Inf[
]-1,2[
```

The left delimiter can be `[` (inclusive) or `]` (exclusive). The right delimiter can be `[` (exclusive) or `]` (inclusive). Beside numbers, you can use `-Inf` and `+Inf` for the infinite.

Forcing the Translator Locale

When translating a message, the Translator uses the specified locale or the `fallback` locale if necessary. You can also manually specify the locale to use for translation:

Listing 70-18
```
$translator->trans(
    'Symfony is great',
    array(),
    'messages',
    'fr_FR'
);

```

4. http://en.wikipedia.org/wiki/Interval_(mathematics)#Notations_for_intervals
5. http://api.symfony.com/2.3/Symfony/Component/Translation/Interval.html

```
$translator->transChoice(
    '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10),
    'messages',
    'fr_FR'
);
```

Chapter 71

Adding Custom Format Support

Sometimes, you need to deal with custom formats for translation files. The Translation component is flexible enough to support this. Just create a loader (to load translations) and, optionally, a dumper (to dump translations).

Imagine that you have a custom format where translation messages are defined using one line for each translation and parentheses to wrap the key and the message. A translation file would look like this:

Listing 71-1
```
(welcome)(accueil)
(goodbye)(au revoir)
(hello)(bonjour)
```

Creating a Custom Loader

To define a custom loader that is able to read these kinds of files, you must create a new class that implements the *LoaderInterface*[1]. The *load()*[2] method will get a filename and parse it into an array. Then, it will create the catalog that will be returned:

Listing 71-2
```
use Symfony\Component\Translation\MessageCatalogue;
use Symfony\Component\Translation\Loader\LoaderInterface;

class MyFormatLoader implements LoaderInterface
{
    public function load($resource, $locale, $domain = 'messages')
    {
        $messages = array();
        $lines = file($resource);

        foreach ($lines as $line) {
            if (preg_match('/\(([^\)]+)\)\(([^\)]+)\)/', $line, $matches)) {
                $messages[$matches[1]] = $matches[2];
```

1. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/LoaderInterface.html
2. http://api.symfony.com/2.3/Symfony/Component/Translation/Loader/LoaderInterface.html#load()

```
            }
        }

        $catalogue = new MessageCatalogue($locale);
        $catalogue->add($messages, $domain);

        return $catalogue;
    }

}
```

Once created, it can be used as any other loader:

Listing 71-3
```
use Symfony\Component\Translation\Translator;

$translator = new Translator('fr_FR');
$translator->addLoader('my_format', new MyFormatLoader());

$translator->addResource('my_format', __DIR__.'/translations/messages.txt', 'fr_FR');

echo $translator->trans('welcome');
```

It will print *"accueil"*.

Creating a Custom Dumper

It is also possible to create a custom dumper for your format, which is useful when using the extraction commands. To do so, a new class implementing the *DumperInterface*[3] must be created. To write the dump contents into a file, extending the *FileDumper*[4] class will save a few lines:

Listing 71-4
```
use Symfony\Component\Translation\MessageCatalogue;
use Symfony\Component\Translation\Dumper\FileDumper;

class MyFormatDumper extends FileDumper
{
    protected function format(MessageCatalogue $messages, $domain = 'messages')
    {
        $output = '';

        foreach ($messages->all($domain) as $source => $target) {
            $output .= sprintf("(%s)(%s)\n", $source, $target);
        }

        return $output;
    }

    protected function getExtension()
    {
        return 'txt';
    }
}
```

3. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/DumperInterface.html
4. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/FileDumper.html

The *format()*[5] method creates the output string, that will be used by the *dump()*[6] method of the FileDumper class to create the file. The dumper can be used like any other built-in dumper. In the following example, the translation messages defined in the YAML file are dumped into a text file with the custom format:

Listing 71-5
```php
use Symfony\Component\Translation\Loader\YamlFileLoader;

$loader = new YamlFileLoader();
$catalogue = $loader->load(__DIR__ . '/translations/messages.fr_FR.yml' , 'fr_FR');

$dumper = new MyFormatDumper();
$dumper->dump($catalogue, array('path' => __DIR__.'/dumps'));
```

5. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/FileDumper.html#format()
6. http://api.symfony.com/2.3/Symfony/Component/Translation/Dumper/FileDumper.html#dump()

Chapter 72
The Yaml Component

> The Yaml component loads and dumps YAML files.

What is It?

The Symfony Yaml component parses YAML strings to convert them to PHP arrays. It is also able to convert PHP arrays to YAML strings.

YAML[1], *YAML Ain't Markup Language*, is a human friendly data serialization standard for all programming languages. YAML is a great format for your configuration files. YAML files are as expressive as XML files and as readable as INI files.

The Symfony Yaml Component implements a selected subset of features defined in the *YAML 1.2 version specification*[2].

Learn more about the Yaml component in the *The YAML Format* article.

Installation

You can install the component in 2 different ways:

- *Install it via Composer* (`symfony/yaml` on *Packagist*[3]);
- Use the official Git repository (*https://github.com/symfony/Yaml*[4]).

1. `http://yaml.org/`
2. `http://yaml.org/spec/1.2/spec.html`
3. `https://packagist.org/packages/symfony/yaml`
4. `https://github.com/symfony/Yaml`

Why?

Fast

One of the goals of Symfony Yaml is to find the right balance between speed and features. It supports just the needed features to handle configuration files. Notable lacking features are: document directives, multi-line quoted messages, compact block collections and multi-document files.

Real Parser

It sports a real parser and is able to parse a large subset of the YAML specification, for all your configuration needs. It also means that the parser is pretty robust, easy to understand, and simple enough to extend.

Clear Error Messages

Whenever you have a syntax problem with your YAML files, the library outputs a helpful message with the filename and the line number where the problem occurred. It eases the debugging a lot.

Dump Support

It is also able to dump PHP arrays to YAML with object support, and inline level configuration for pretty outputs.

Types Support

It supports most of the YAML built-in types like dates, integers, octals, booleans, and much more...

Full Merge Key Support

Full support for references, aliases, and full merge key. Don't repeat yourself by referencing common configuration bits.

Using the Symfony YAML Component

The Symfony Yaml component is very simple and consists of two main classes: one parses YAML strings (*Parser*[5]), and the other dumps a PHP array to a YAML string (*Dumper*[6]).

On top of these two classes, the *Yaml*[7] class acts as a thin wrapper that simplifies common uses.

Reading YAML Files

The *parse()*[8] method parses a YAML string and converts it to a PHP array:

Listing 72-1
```
1 use Symfony\Component\Yaml\Parser;
2
3 $yaml = new Parser();
```

5. http://api.symfony.com/2.3/Symfony/Component/Yaml/Parser.html
6. http://api.symfony.com/2.3/Symfony/Component/Yaml/Dumper.html
7. http://api.symfony.com/2.3/Symfony/Component/Yaml/Yaml.html
8. http://api.symfony.com/2.3/Symfony/Component/Yaml/Parser.html#parse()

```
4
5 $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
```

If an error occurs during parsing, the parser throws a *ParseException*[9] exception indicating the error type and the line in the original YAML string where the error occurred:

Listing 72-2
```
1 use Symfony\Component\Yaml\Exception\ParseException;
2
3 try {
4     $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
5 } catch (ParseException $e) {
6     printf("Unable to parse the YAML string: %s", $e->getMessage());
7 }
```

As the parser is re-entrant, you can use the same parser object to load different YAML strings.

It may also be convenient to use the *parse()*[10] wrapper method:

Listing 72-3
```
1 use Symfony\Component\Yaml\Yaml;
2
3 $yaml = Yaml::parse(file_get_contents('/path/to/file.yml'));
```

The *parse()*[11] static method takes a YAML string or a file containing YAML. Internally, it calls the *parse()*[12] method, but enhances the error if something goes wrong by adding the filename to the message.

Because it is currently possible to pass a filename to this method, you must validate the input first. Passing a filename is deprecated in Symfony 2.2, and will be removed in Symfony 3.0.

Writing YAML Files

The *dump()*[13] method dumps any PHP array to its YAML representation:

Listing 72-4
```
1  use Symfony\Component\Yaml\Dumper;
2
3  $array = array(
4      'foo' => 'bar',
5      'bar' => array('foo' => 'bar', 'bar' => 'baz'),
6  );
7
8  $dumper = new Dumper();
9
10 $yaml = $dumper->dump($array);
```

9. http://api.symfony.com/2.3/Symfony/Component/Yaml/Exception/ParseException.html
10. http://api.symfony.com/2.3/Symfony/Component/Yaml/Yaml.html#parse()
11. http://api.symfony.com/2.3/Symfony/Component/Yaml/Yaml.html#parse()
12. http://api.symfony.com/2.3/Symfony/Component/Yaml/Parser.html#parse()
13. http://api.symfony.com/2.3/Symfony/Component/Yaml/Dumper.html#dump()

```
11
12 file_put_contents('/path/to/file.yml', $yaml);
```

Of course, the Symfony Yaml dumper is not able to dump resources. Also, even if the dumper is able to dump PHP objects, it is considered to be a not supported feature.

If an error occurs during the dump, the parser throws a *DumpException*[14] exception.

If you only need to dump one array, you can use the *dump()*[15] static method shortcut:

Listing 72-5
```
1 use Symfony\Component\Yaml\Yaml;
2
3 $yaml = Yaml::dump($array, $inline);
```

The YAML format supports two kind of representation for arrays, the expanded one, and the inline one. By default, the dumper uses the inline representation:

Listing 72-6
```
1 { foo: bar, bar: { foo: bar, bar: baz } }
```

The second argument of the *dump()*[16] method customizes the level at which the output switches from the expanded representation to the inline one:

Listing 72-7
```
1 echo $dumper->dump($array, 1);
```

Listing 72-8
```
1 foo: bar
2 bar: { foo: bar, bar: baz }
```

Listing 72-9
```
1 echo $dumper->dump($array, 2);
```

Listing 72-10
```
1 foo: bar
2 bar:
3     foo: bar
4     bar: baz
```

14. http://api.symfony.com/2.3/Symfony/Component/Yaml/Exception/DumpException.html
15. http://api.symfony.com/2.3/Symfony/Component/Yaml/Yaml.html#dump()
16. http://api.symfony.com/2.3/Symfony/Component/Yaml/Dumper.html#dump()

Chapter 73
The YAML Format

According to the official *YAML*[1] website, YAML is "a human friendly data serialization standard for all programming languages".

Even if the YAML format can describe complex nested data structure, this chapter only describes the minimum set of features needed to use YAML as a configuration file format.

YAML is a simple language that describes data. As PHP, it has a syntax for simple types like strings, booleans, floats, or integers. But unlike PHP, it makes a difference between arrays (sequences) and hashes (mappings).

Scalars

The syntax for scalars is similar to the PHP syntax.

Strings

Strings in YAML can be wrapped both in single and double quotes. In some cases, they can also be unquoted:

Listing 73-1
```
A string in YAML

'A singled-quoted string in YAML'

"A double-quoted string in YAML"
```

Quoted styles are useful when a string starts or end with one or more relevant spaces, because unquoted strings are trimmed on both end when parsing their contents. Quotes are required when the string contains special or reserved characters.

When using single-quoted strings, any single quote `'` inside its contents must be doubled to escape it:

Listing 73-2
```
'A single quote '' inside a single-quoted string'
```

1. http://yaml.org/

Strings containing any of the following characters must be quoted. Although you can use double quotes, for these characters it is more convenient to use single quotes, which avoids having to escape any backslash `\`:

- `:`, `{`, `}`, `[`, `]`, `,`, `&`, `*`, `#`, `?`, `|`, `-`, `<`, `>`, `=`, `!`, `%`, `@`, `\``

The double-quoted style provides a way to express arbitrary strings, by using `\` to escape characters and sequences. For instance, it is very useful when you need to embed a `\n` or a Unicode character in a string.

Listing 73-3
```
"A double-quoted string in YAML\n"
```

If the string contains any of the following control characters, it must be escaped with double quotes:

- `\0`, `\x01`, `\x02`, `\x03`, `\x04`, `\x05`, `\x06`, `\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\x0e`, `\x0f`, `\x10`, `\x11`, `\x12`, `\x13`, `\x14`, `\x15`, `\x16`, `\x17`, `\x18`, `\x19`, `\x1a`, `\e`, `\x1c`, `\x1d`, `\x1e`, `\x1f`, `\N`, `\_`, `\L`, `\P`

Finally, there are other cases when the strings must be quoted, no matter if you're using single or double quotes:

- When the string is `true` or `false` (otherwise, it would be treated as a boolean value);
- When the string is `null` or `~` (otherwise, it would be considered as a `null` value);
- When the string looks like a number, such as integers (e.g. `2`, `14`, etc.), floats (e.g. `2.6`, `14.9`) and exponential numbers (e.g. `12e7`, etc.) (otherwise, it would be treated as a numeric value);
- When the string looks like a date (e.g. `2014-12-31`) (otherwise it would be automatically converted into a Unix timestamp).

When a string contains line breaks, you can use the literal style, indicated by the pipe (`|`), to indicate that the string will span several lines. In literals, newlines are preserved:

Listing 73-4
```
|
  \/ /| |\/| |
  / / | |  | |__
```

Alternatively, strings can be written with the folded style, denoted by `>`, where each line break is replaced by a space:

Listing 73-5
```
>
  This is a very long sentence
  that spans several lines in the YAML
  but which will be rendered as a string
  without carriage returns.
```

Notice the two spaces before each line in the previous examples. They won't appear in the resulting PHP strings.

Numbers

Listing 73-6
```
# an integer
12
```

Listing 73-7

```
# an octal
014
```

Listing 73-8
```
# an hexadecimal
0xC
```

Listing 73-9
```
# a float
13.4
```

Listing 73-10
```
# an exponential number
1.2e+34
```

Listing 73-11
```
# infinity
.inf
```

Nulls

Nulls in YAML can be expressed with `null` or `~`.

Booleans

Booleans in YAML are expressed with `true` and `false`.

Dates

YAML uses the ISO-8601 standard to express dates:

Listing 73-12
```
2001-12-14t21:59:43.10-05:00
```

Listing 73-13
```
# simple date
2002-12-14
```

Collections

A YAML file is rarely used to describe a simple scalar. Most of the time, it describes a collection. A collection can be a sequence or a mapping of elements. Both sequences and mappings are converted to PHP arrays.

Sequences use a dash followed by a space:

Listing 73-14
```
- PHP
- Perl
- Python
```

The previous YAML file is equivalent to the following PHP code:

Listing 73-15

```
1 array('PHP', 'Perl', 'Python');
```

Mappings use a colon followed by a space (`: `) to mark each key/value pair:

Listing 73-16
```
1 PHP: 5.2
2 MySQL: 5.1
3 Apache: 2.2.20
```

which is equivalent to this PHP code:

Listing 73-17
```
1 array('PHP' => 5.2, 'MySQL' => 5.1, 'Apache' => '2.2.20');
```

In a mapping, a key can be any valid scalar.

The number of spaces between the colon and the value does not matter:

Listing 73-18
```
1 PHP:    5.2
2 MySQL:  5.1
3 Apache: 2.2.20
```

YAML uses indentation with one or more spaces to describe nested collections:

Listing 73-19
```
1 "symfony 1.0":
2   PHP:    5.0
3   Propel: 1.2
4 "symfony 1.2":
5   PHP:    5.2
6   Propel: 1.3
```

The following YAML is equivalent to the following PHP code:

Listing 73-20
```
1  array(
2      'symfony 1.0' => array(
3          'PHP'    => 5.0,
4          'Propel' => 1.2,
5      ),
6      'symfony 1.2' => array(
7          'PHP'    => 5.2,
8          'Propel' => 1.3,
9      ),
10 );
```

There is one important thing you need to remember when using indentation in a YAML file: *Indentation must be done with one or more spaces, but never with tabulations.*

You can nest sequences and mappings as you like:

Listing 73-21
```
1 'Chapter 1':
2   - Introduction
3   - Event Types
4 'Chapter 2':
```

```
5     - Introduction
6     - Helpers
```

YAML can also use flow styles for collections, using explicit indicators rather than indentation to denote scope.

A sequence can be written as a comma separated list within square brackets (`[]`):

Listing 73-22
```
1  [PHP, Perl, Python]
```

A mapping can be written as a comma separated list of key/values within curly braces (`{}`):

Listing 73-23
```
1  { PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

You can mix and match styles to achieve a better readability:

Listing 73-24
```
1  'Chapter 1': [Introduction, Event Types]
2  'Chapter 2': [Introduction, Helpers]
```

Listing 73-25
```
1  "symfony 1.0": { PHP: 5.0, Propel: 1.2 }
2  "symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Comments

Comments can be added in YAML by prefixing them with a hash mark (`#`):

Listing 73-26
```
1  # Comment on a line
2  "symfony 1.0": { PHP: 5.0, Propel: 1.2 } # Comment at the end of a line
3  "symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Comments are simply ignored by the YAML parser and do not need to be indented according to the current level of nesting in a collection.