Symfony

The Components Book

for Symfony 2.0

generated on November 25, 2013

Contents at a Glance

Chapter 1

How to Install and Use the Symfony2 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. 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 *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. Create a new file called `composer.json` and paste the following into it:

Listing 1-1
```
{
    "require": {
        "symfony/finder": "2.1.*"
    }
}
```

If you already have a `composer.json` file, just add this line to it. You may also need to adjust the version (e.g. `2.1.1` or `2.2.*`).

You can research the component names and versions at *packagist.org*[1].

3. Download the vendor libraries and generate the `vendor/autoload.php` file:

Listing 1-2
```
$ php composer.phar install
```

4. Write your code:

1. https://packagist.org/

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:

Listing 1-3
```php
// File: src/script.php

require_once '../vendor/autoload.php';

use Symfony\Component\Finder\Finder;

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

// ...
```

If you want to use all of the Symfony2 Components, then instead of adding them one by one:

Listing 1-4
```json
{
    "require": {
        "symfony/finder": "2.1.*",
        "symfony/dom-crawler": "2.1.*",
        "symfony/css-selector": "2.1.*"
    }
}
```

you can use:

Listing 1-5
```json
{
    "require": {
        "symfony/symfony": "2.1.*"
    }
}
```

This will include the Bundle and Bridge libraries, which you 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 loads your project classes automatically if they follow some standard PHP conventions.

Whenever you use an undefined class, PHP uses the autoloading mechanism to delegate the loading of a file defining the class. Symfony2 provides a "universal" autoloader, which is able to load classes from files that implement one of the following conventions:

- The technical interoperability *standards*[1] for PHP 5.3 namespaces and class names;
- The *PEAR*[2] naming convention for classes.

If your classes and the third-party libraries you use for your project follow these standards, the Symfony2 autoloader is the only autoloader you will ever need.

Installation

You can install the component in many different ways:

- Use the official Git repository (*https://github.com/symfony/ClassLoader*[3]);
- *Install it via Composer* (`symfony/class-loader` on *Packagist*[4]).

Usage

Registering the *`UniversalClassLoader`*[5] autoloader is straightforward:

Listing 2-1

1. http://symfony.com/PSR0
2. http://pear.php.net/manual/en/standards.php
3. https://github.com/symfony/ClassLoader
4. https://packagist.org/packages/symfony/class-loader
5. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html

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

use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();

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

$loader->register();
```

For minor performance gains class paths can be cached in memory using APC by registering the *ApcUniversalClassLoader*[6]:

Listing 2-2
```php
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('apc.prefix.');
$loader->register();
```

The autoloader is useful only if you add some libraries to autoload.

The autoloader is automatically registered in a Symfony2 application (see `app/autoload.php`).

If the classes to autoload use namespaces, use the *registerNamespace()*[7] or *registerNamespaces()*[8] methods:

Listing 2-3
```php
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/src');

$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/src',
));

$loader->register();
```

For classes that follow the PEAR naming convention, use the *registerPrefix()*[9] or *registerPrefixes()*[10] methods:

Listing 2-4
```php
$loader->registerPrefix('Twig_', __DIR__.'/vendor/twig/lib');

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

6. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/ApcUniversalClassLoader.html
7. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerNamespace()
8. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerNamespaces()
9. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerPrefix()
10. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerPrefixes()

```
7
8 $loader->register();
```

Some libraries also require their root path be registered in the PHP include path (`set_include_path()`).

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

Listing 2-5
```
1 $loader->registerNamespaces(array(
2     'Doctrine\\Common'           => __DIR__.'/vendor/doctrine-common/lib',
3     'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine-migrations/lib',
4     'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine-dbal/lib',
5     'Doctrine'                   => __DIR__.'/vendor/doctrine/lib',
6 ));
7
8 $loader->register();
```

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, and it will then fallback to the default `Doctrine` directory (the last one configured) if not found, before giving up. The order of the registrations is significant in this case.

Chapter 3

The Config Component

Introduction

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 many different ways:

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

Sections

- *Loading resources*
- *Caching based on resources*
- *Defining and processing configuration values*

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

Chapter 4
Loading resources

Locating resources

Loading the configuration normally starts with a search for resources – in most cases: files. This can be done with the *FileLocator*[1]:

Listing 4-1
```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.

Resource loaders

For each type of resource (Yaml, XML, annotation, etc.) a loader must be defined. Each loader should implement *LoaderInterface*[2] or extend the abstract *FileLoader*[3] class, which allows for recursively importing other resources:

Listing 4-2
```php
use Symfony\Component\Config\Loader\FileLoader;
use Symfony\Component\Yaml\Yaml;

```

1. http://api.symfony.com/2.0/Symfony/Component/Config/FileLocator.html
2. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/LoaderInterface.html
3. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/FileLoader.html

```php
class YamlUserLoader extends FileLoader
{
    public function load($resource, $type = null)
    {
        $configValues = Yaml::parse($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*[4] 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*[5] makes use of the *LoaderResolver*[6]. When it is asked to load a resource, it delegates this question to the *LoaderResolver*[7]. In case the resolver has found a suitable loader, this loader will be asked to load the resource:

Listing 4-3

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

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

$delegatingLoader->load(__DIR__.'/users.yml');
/*
The YamlUserLoader will be used to load this resource,
since it supports files with a "yml" extension
*/
```

4. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/LoaderResolver.html
5. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/DelegatingLoader.html
6. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/LoaderResolver.html
7. http://api.symfony.com/2.0/Symfony/Component/Config/Loader/LoaderResolver.html

Chapter 5
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 5-1

```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/2.0/Symfony/Component/Config/ConfigCache.html

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

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

Chapter 6
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 6-1

```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 6-2
```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 6-3
```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 type are available for:

1. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
2. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
3. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/ConfigurationInterface.html

- scalar
- boolean
- array
- variable (no validation)

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

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 6-4
```php
$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 6-5
```php
$rootNode
    ->children()
        ->arrayNode('connections')
            ->prototype('array')
            ->children()
                ->scalarNode('driver')->end()
                ->scalarNode('host')->end()
                ->scalarNode('username')->end()
                ->scalarNode('password')->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 6-6
```
$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 6-7
```
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 6-8
```
$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()
;
```

Merging options

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

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

For all nodes:

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

Appending sections

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

Listing 6-9

```
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`. 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 6-10

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

and in XML:

Listing 6-11
```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 6-12
```php
$rootNode
    ->fixXmlConfig('extension')
    ->children()
        ->arrayNode('extensions')
            ->prototype('scalar')->end()
        ->end()
    ->end()
;
```

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

Listing 6-13
```php
$rootNode
    ->fixXmlConfig('child', 'children')
    ->children()
        ->arrayNode('children')
    ->end()
;
```

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

Listing 6-14
```xml
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 6-15
```xml
<connection>default</connection>
```

By default `connection` would be an array in the first case and a string in the second making it difficult to validate. You can ensure it is always an array with 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 6-16
```yaml
connection:
    name: my_mysql_connection
    host: localhost
    driver: mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 6-17
```
connection: my_mysql_connection
```

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

Listing 6-18
```
$rootNode
    ->children()
        ->arrayNode('connection')
            ->beforeNormalization()
            ->ifString()
                ->then(function($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*[4]. 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 6-19
```
$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:

4. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/Builder/ExprBuilder.html

- 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*[5] uses the tree as it was built using the *TreeBuilder*[6] 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 6-20

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

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

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

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

5. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/Processor.html
6. http://api.symfony.com/2.0/Symfony/Component/Config/Definition/Builder/TreeBuilder.html

Chapter 7
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 many different ways:

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

Windows does not support ANSI colors by default so the Console Component detects and disables colors where Windows does not have support. However, if Windows is not configured with an ANSI driver and your console commands invoke other scripts which emit ANSI color sequences, they will be shown as raw escape characters.

To enable ANSI colour support for Windows, please install *ANSICON*[3].

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 7-1
```
1  namespace Acme\DemoBundle\Command;
2
```

1. https://github.com/symfony/Console
2. https://packagist.org/packages/symfony/console
3. http://adoxa.3eeweb.com/ansicon/

```php
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 GreetCommand extends Command
{
    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 7-2

```php
#!/usr/bin/env php
<?php
// app/console

use Acme\DemoBundle\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 7-3
```
$ app/console demo:greet Fabien
```

This will print the following to the command line:

Listing 7-4
```
Hello Fabien
```

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

Listing 7-5
```
$ app/console demo:greet Fabien --yell
```

This prints:

Listing 7-6
```
HELLO FABIEN
```

Coloring the Output

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

Listing 7-7
```
// green text
$output->writeln('<info>foo</info>');

// yellow text
$output->writeln('<comment>foo</comment>');

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

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

It is possible to define your own styles using the class *OutputFormatterStyle*[4]:

Listing 7-8
```
$style = new OutputFormatterStyle('red', 'yellow', array('bold', 'blink'));
$output->getFormatter()->setStyle('fire', $style);
$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 7-9
```
// 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>');
```

4. http://api.symfony.com/2.0/Symfony/Component/Console/Formatter/OutputFormatterStyle.html

Verbosity Levels

The console has 3 levels of verbosity. These are defined in the *OutputInterface*[5]:

Option	Value
OutputInterface::VERBOSITY_QUIET	Do not output any messages
OutputInterface::VERBOSITY_NORMAL	The default verbosity level
OutputInterface::VERBOSITY_VERBOSE	Increased verbosity of 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 is used.

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

Listing 7-10
```php
if (OutputInterface::VERBOSITY_VERBOSE === $output->getVerbosity()) {
    $output->writeln(...);
}
```

When the quiet level is used, all output is suppressed as the default *SymfonyComponentConsoleOutput::write*[6] 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 7-11
```php
$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 7-12

5. http://api.symfony.com/2.0/Symfony/Component/Console/Output/OutputInterface.html
6. http://api.symfony.com/2.0/Symfony/Component/Console/Output.html#write()

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

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

Listing 7-13
```
$ app/console demo:greet Fabien
$ app/console demo:greet Fabien Potencier
```

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`).

It is also possible to make an option *optionally* accept a value (so that `--yell` or `yell=loud` work). Options can also be configured to accept an array of values.

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 7-14
```php
$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 7-15
```php
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 7-16
```
$ app/console demo:greet Fabien
$ app/console 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 7-17
```
$ app/console demo:greet Fabien --iterations=5 --yell
$ app/console 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 7-18
```php
$this
    // ...
    ->addOption(
        'iterations',
        null,
        InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY,
        'How many times should the message be printed?',
        1
    );
```

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

Testing Commands

Symfony2 provides several tools to help you test your commands. The most useful one is the *CommandTester*[7] class. It uses special input and output classes to ease testing without a real console:

Listing 7-19
```php
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;
use Acme\DemoBundle\Command\GreetCommand;

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

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

7. http://api.symfony.com/2.0/Symfony/Component/Console/Tester/CommandTester.html

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

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

        // ...
    }
}
```

The *getDisplay()*[8] 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()*[9] method:

Listing 7-20

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

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')
        );

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

You can also test a whole console application by using *ApplicationTester*[10].

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:

8. http://api.symfony.com/2.0/Symfony/Component/Console/Tester/CommandTester.html#getDisplay()
9. http://api.symfony.com/2.0/Symfony/Component/Console/Tester/CommandTester.html#execute()
10. http://api.symfony.com/2.0/Symfony/Component/Console/Tester/ApplicationTester.html

Listing 7-21

```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()*[11] the command you want to execute by passing the command name.

Then, you need to create a new *ArrayInput*[12] 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*

11. http://api.symfony.com/2.0/Symfony/Component/Console/Application.html#find()
12. http://api.symfony.com/2.0/Symfony/Component/Console/Input/ArrayInput.html

Chapter 8

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 `app/console` to run at the cli:

Listing 8-1
```php
#!/usr/bin/env php
# app/console
<?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 8-2
```
$ php app/console list
```

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

Listing 8-3
```
$ php app/console
```

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

Listing 8-4
```
$ php app/console help list
```

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

Listing 8-5
```
$ php app/console help
```

Global Options

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

Listing 8-6
```
$ php app/console list --help
$ php app/console list -h
```

You can suppress output with:

Listing 8-7
```
$ php app/console list --quiet
$ php app/console list -q
```

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

Listing 8-8
```
$ php app/console list --verbose
$ php app/console list -v
```

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

Listing 8-9
```
$application = new Application('Acme Console Application', '1.2');
```

then you can use:

Listing 8-10
```
$ php app/console list --version
$ php app/console list -V
```

to get this information output:

Listing 8-11
```
Acme Console Application version 1.2
```

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

Listing 8-12
```
console tool
```

You can force turning on ANSI output coloring with:

Listing 8-13
```
$ php app/console list --ansi
```

or turn it off with:

Listing 8-14
```
$ php app/console list --no-ansi
```

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

Listing 8-15
```
$ php app/console list --no-interaction
$ php app/console 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 8-16
```
$ php app/console 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 8-17
```
$ php app/console 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 9
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 9-1

```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 9-2

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

Chapter 10
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 10-1
```php
$dialog = $this->getHelperSet()->get('dialog');
```

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

Asking the User for confirmation

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

Listing 10-2
```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 in any other case. The third argument to `askConfirmation` is the default value to return if the user doesn't enter any input.

1. http://api.symfony.com/2.0/Symfony/Component/Console/Helper/DialogHelper.html
2. http://api.symfony.com/2.0/Symfony/Component/Console/Command/Command.html#getHelperSet()
3. http://api.symfony.com/2.0/Symfony/Component/Console/Output/OutputInterface.html

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 10-3
```
// ...
$bundle = $dialog->ask(
    $output,
    'Please enter the name of the bundle',
    'AcmeDemoBundle'
);
```

The user will be asked "Please enter the name of the bundle". She can type some name which will be returned by the `ask` method. If she leaves it empty, the default value (`AcmeDemoBundle` here) is returned.

Validating the Answer

You can even validate the answer. For instance, in the last example you asked for the bundle name. Following the Symfony2 naming conventions, it should be suffixed with `Bundle`. You can validate that by using the *`askAndValidate()`*[4] method:

Listing 10-4
```
// ...
$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 10-5
```
askAndValidate(
    OutputInterface $output,
    string|array $question,
    callback $validator,
    integer $attempts = false,
    string $default = 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, which is given in the last argument. Using `false` means the amount of attempts

4. http://api.symfony.com/2.0/Symfony/Component/Console/Helper/DialogHelper.html#askAndValidate()

is infinite. The user will be asked as long as he provides an invalid answer and will only be able to proceed if her input is valid.

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 10-6

```php
use Symfony\Component\Console\Helper\DialogHelper;
use Symfony\Component\Console\Helper\HelperSet;

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

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

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

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

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

    return $stream;
}
```

By setting the inputStream 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.

Chapter 11
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 11-1
```php
$formatter = $this->getHelperSet()->get('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 11-2
```
[SomeSection] Here is some message related to that section
```

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

Listing 11-3
```php
$formattedLine = $formatter->formatSection(
    'SomeSection',
    'Here is some message related to that section'
);
$output->writeln($formattedLine);
```

1. http://api.symfony.com/2.0/Symfony/Component/Console/Helper/FormatterHelper.html
2. http://api.symfony.com/2.0/Symfony/Component/Console/Command/Command.html#getHelperSet()
3. http://api.symfony.com/2.0/Symfony/Component/Console/Output/OutputInterface.html#writeln()
4. http://api.symfony.com/2.0/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 11-4
```
1 $errorMessages = array('Error!', 'Something went wrong');
2 $formattedBlock = $formatter->formatBlock($errorMessages, 'error');
3 $output->writeln($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/2.0/Symfony/Component/Console/Helper/FormatterHelper.html#formatBlock()
6. http://api.symfony.com/2.0/Symfony/Component/Console/Helper/FormatterHelper.html#formatBlock()

Chapter 12
The CssSelector Component

The CssSelector Component converts CSS selectors to XPath expressions.

Installation

You can install the component in several different ways:

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

Usage

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

The `CssSelector` component

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

Listing 12-1
```php
use Symfony\Component\CssSelector\CssSelector;

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

This gives the following output:

Listing 12-2
```
descendant-or-self::div[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: `:enabled`, `:disabled`, `:indeterminate` (however, `: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:

- `:lang(language)`
- `root`
- `*: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/2.0/Symfony/Component/DomCrawler/Crawler.html#filter()

Chapter 13
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 many different ways:

- Use the official Git repository (*https://github.com/symfony/DomCrawler*[1]);
- *Install it via Composer* (`symfony/dom-crawler` on *Packagist*[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 13-1

```
1  use Symfony\Component\DomCrawler\Crawler;
2
3  $html = <<<'HTML'
4  <!DOCTYPE html>
```

1. https://github.com/symfony/DomCrawler
2. https://packagist.org/packages/symfony/dom-crawler
3. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Crawler.html
4. http://php.net/manual/en/class.splobjectstorage.php
5. http://php.net/manual/en/class.domelement.php

```
<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.

Node Filtering

Using XPath expressions is really easy:

Listing 13-2
```
$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 13-3
```
$crawler = $crawler->filter('body > p');
```

Anonymous function can be used to filter with more complex criteria:

Listing 13-4
```
$crawler = $crawler->filter('body > p')->reduce(function ($node, $i) {
    // filter even nodes
    return ($i % 2) == 0;
});
```

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:

6. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Link.html
7. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Form.html
8. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Crawler.html

Listing 13-5
```
$crawler->filter('body > p')->eq(0);
```

Get the first or last node of the current selection:

Listing 13-6
```
$crawler->filter('body > p')->first();
$crawler->filter('body > p')->last();
```

Get the nodes of the same level as the current selection:

Listing 13-7
```
$crawler->filter('body > p')->siblings();
```

Get the same level nodes after or before the current selection:

Listing 13-8
```
$crawler->filter('body > p')->nextAll();
$crawler->filter('body > p')->previousAll();
```

Get all the child or parent nodes:

Listing 13-9
```
$crawler->filter('body')->children();
$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 13-10
```
$message = $crawler->filterXPath('//body/p')->text();
```

Access the attribute value of the first node of the current selection:

Listing 13-11
```
$class = $crawler->filterXPath('//body/p')->attr('class');
```

Extract attribute and/or node values from the list of nodes:

Listing 13-12
```
$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:

9. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Crawler.html

Listing 13-13
```php
$nodeValues = $crawler->filter('p')->each(function ($node, $i) {
    return $node->text();
});
```

The anonymous function receives the position and the node 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 13-14
```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()*[10] 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*[11], *DOMNodeList*[12] and *DOMNode*[13] objects:

Listing 13-15
```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);
```

10. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Crawler.html#addHTMLContent()
11. http://php.net/manual/en/class.domdocument.php
12. http://php.net/manual/en/class.domnodelist.php
13. http://php.net/manual/en/class.domnode.php

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`*[14] objects, you can use any method or property available on *`DOMElement`*[15], *`DOMNode`*[16] or *`DOMDocument`*[17]. For example, you could get the HTML of a `Crawler` with something like this:

Listing 13-16
```php
$html = '';

foreach ($crawler as $domElement) {
    $html .= $domElement->ownerDocument->saveHTML($domElement);
}
```

Form and Link support

Special treatment is given to links and forms inside the DOM tree.

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`*[18] object:

Listing 13-17
```php
$linksCrawler = $crawler->selectLink('Go elsewhere...');
$link = $linksCrawler->link();

// or do this all at once
$link = $crawler->selectLink('Go elsewhere...')->link();
```

The *`Link`*[19] object has several useful methods to get more information about the selected link itself:

Listing 13-18
```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`)

14. http://php.net/manual/en/class.domelement.php
15. http://php.net/manual/en/class.domelement.php
16. http://php.net/manual/en/class.domnode.php
17. http://php.net/manual/en/class.domdocument.php
18. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Link.html
19. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Link.html

with the given text. This method is especially useful because you can use it to return a *Form*[20] object that represents the form that the button lives in:

Listing 13-19
```php
$form = $crawler->selectButton('validate')->form();

// or "fill" the form fields with data
$form = $crawler->selectButton('validate')->form(array(
    'name' => 'Ryan',
));
```

The *Form*[21] object has lots of very useful methods for working with forms:

Listing 13-20
```php
$uri = $form->getUri();

$method = $form->getMethod();
```

The *getUri()*[22] 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 13-21
```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 13-22
```php
<form>
    <input name="multi[]" />
    <input name="multi[]" />
    <input name="multi[dimensional]" />
</form>
```

Pass an array of values:

Listing 13-23
```php
// 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'
)));
```

20. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Form.html
21. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Form.html
22. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Form.html#getUri()

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 13-24
```php
$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 or checkboxes
$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 13-25
```php
$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 13-26
```php
$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*[23]. Goutte understands the Symfony Crawler object and can use it to submit forms directly:

Listing 13-27
```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);
```

23. https://github.com/fabpot/goutte

Chapter 14
The Dependency Injection Component

> The Dependency Injection 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 many different ways:

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

Basic Usage

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

Listing 14-1
```php
class Mailer
{
    private $transport;

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

    // ...
}
```

You can register this in the container as a service:

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

Listing 14-2
```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 14-3
```php
class Mailer
{
    private $transport;

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

    // ...
}
```

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

Listing 14-4
```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 14-5
```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 14-6
```php
class NewsletterManager
{
    private $mailer;

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

```

```
    // ...
}
```

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

Listing 14-7
```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;

$container = new ContainerBuilder();

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

$container
    ->register('newsletter_manager', 'NewsletterManager')
    ->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 14-8
```
class NewsletterManager
{
    private $mailer;

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

    // ...
}
```

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 14-9
```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;

$container = new ContainerBuilder();

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

$container
    ->register('newsletter_manager', 'NewsletterManager')
    ->addMethodCall('setMailer', array(new Reference('mailer')));
```

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

Listing 14-10
```
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
```

```
4
5 // ...
6
7 $newsletterManager = $container->get('newsletter_manager');
```

Avoiding Your Code Becoming Dependent on the Container

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

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

Setting Up the Container with Configuration Files

As well as setting up the services using PHP as above you can also use configuration files. This allows you to use XML or Yaml to write the definitions for the services rather than using PHP to define the services as in the above examples. In anything but the smallest applications it make 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 14-11
```
1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2 use Symfony\Component\Config\FileLocator;
3 use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
4
5 $container = new ContainerBuilder();
6 $loader = new XmlFileLoader($container, new FileLocator(__DIR__));
7 $loader->load('services.xml');
```

Loading a YAML config file:

Listing 14-12
```
1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2 use Symfony\Component\Config\FileLocator;
3 use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
4
5 $container = new ContainerBuilder();
6 $loader = new YamlFileLoader($container, new FileLocator(__DIR__));
7 $loader->load('services.yml');
```

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

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

Listing 14-13

```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 14-14

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

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

Chapter 15
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 15-1
```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 15-2
```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 15-3
```php
class NewsletterManager
{
    protected $mailer;

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

    // ...
}
```

Listing 15-4
```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 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 15-5
```php
class NewsletterManager
{
    public $mailer;

    // ...
}
```

Listing 15-6
```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 16
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 16-1
```php
$container->hasParameter('mailer.transport');
```

You can retrieve a parameter set in the container with:

Listing 16-2
```php
$container->getParameter('mailer.transport');
```

and set a parameter in the container with:

Listing 16-3
```php
$container->setParameter('mailer.transport', 'sendmail');
```

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 16-4
```yaml
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 by making it a parameter. This makes it easier to change rather than being tied up and hidden with the service definition:

Listing 16-5
```yaml
parameters:
    mailer.transport: sendmail

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

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

You can also use the parameters in the service definition, for example, making the class of a service a parameter:

Listing 16-6
```yaml
parameters:
    mailer.transport: sendmail
    mailer.class: Mailer

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

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

Listing 16-7
```yaml
arguments: ['http://symfony.com/?foo=%%s&bar=%%d']
```

Array Parameters

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

Listing 16-8
```yaml
# app/config/config.yml
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 16-9
```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">

    <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 works for Yaml configuration. If you're using Yaml, you can import an XML file to take advantage of this functionality:

Listing 16-10
```yaml
# app/config/config.yml
imports:
    - { resource: parameters.xml }
```

Chapter 17
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 17-1
```
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 17-2
```
1  $container->getDefinition($serviceId);
```

or:

Listing 17-3
```
1  $container->findDefinition($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 17-4
```
1  $container->setDefinition($id, $definition);
```

Working with a definition

Creating a new definition

If you need to create a new definition rather than manipulate one retrieved from then 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 17-5
```php
$definition->getClass();
```

and to set a different class:

Listing 17-6
```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 17-7
```php
$definition->getArguments();
```

or to get a single argument by its position:

Listing 17-8
```php
$definition->getArgument($index);
//e.g. $definition->getArguments(0) for the first argument
```

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

Listing 17-9
```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 17-10
```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 17-11
```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 17-12

1. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/Definition.html

```
1 $definition->replaceArguments($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 17-13
```
1 $definition->getMethodCalls();
```

Add a method call with:

Listing 17-14
```
1 $definition->addMethodCall($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 17-15
```
1 $definition->setMethodCalls($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 18
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.

It is compiled by running:

Listing 18-1
```
1  $container->compile();
```

The compile method uses *Compiler Passes* for the compilation. The *Dependency Injection* 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 Dependency Injection 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 18-2
```
1  $container->registerExtension($extension);
```

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

1. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/Compiler/CheckDefinitionValidityPass.html
2. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html

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

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

Listing 18-3
```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 18-4
```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 18-5
```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 18-6

```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 18-7
```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 18-8
```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 18-9
```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

```

```
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 18-10
```php
public function getXsdValidationBasePath()
{
    return __DIR__.'/../Resources/config/';
}

public function getNamespace()
{
    return 'http://www.example.com/symfony/schema/';
}
```

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 18-11
```xml
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_demo="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/
symfony/schema/hello-1.0.xsd">

    <acme_demo:config>
        <acme_demo:foo>fooValue</acme_hello:foo>
        <acme_demo:bar>barValue</acme_demo:bar>
    </acme_demo:config>

</container>
```

In the Symfony2 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 expose a Semantic Configuration for 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 18-12
```php
public function load(array $configs, ContainerBuilder $container)
{
    $configuration = new Configuration();
    $processor = new Processor();
    $config = $processor->processConfiguration($configuration, $configs);

    $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 18-13
```
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 18-14
```
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.

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*[4] 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.

3. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/ContainerBuilder.html#loadFromExtension()
4. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/Compiler/CompilerPassInterface.html

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

Listing 18-15
```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 18-16
```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 18-17
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\PassConfig;

```

```
4 $container = new ContainerBuilder();
5 $container->addCompilerPass(
6     new CustomCompilerPass,
7     PassConfig::TYPE_AFTER_REMOVING
8 );
```

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 18-18

```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2  use Symfony\Component\DependencyInjection\Dumper\PhpDumper;
3
4  $file = __DIR__ .'/cache/container.php';
5
6  if (file_exists($file)) {
7      require_once $file;
8      $container = new ProjectServiceContainer();
9  } else {
10     $container = new ContainerBuilder();
11     // ...
12     $container->compile();
13
14     $dumper = new PhpDumper($container);
15     file_put_contents($file, $dumper->dump());
16 }
```

`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 18-19

```
1  // ...
2  $file = __DIR__ .'/cache/container.php';
3
4  if (file_exists($file)) {
5      require_once $file;
6      $container = new MyCachedContainer();
7  } else {
8      $container = new ContainerBuilder();
9      // ...
10     $container->compile();
11
12     $dumper = new PhpDumper($container);
13     file_put_contents(
14         $file,
15         $dumper->dump(array('class' => 'MyCachedContainer'))
16     );
17 }
```

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

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

Listing 18-20
```php
// ...

// 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 18-21
```php
// ...

// 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')),
```

```
17          $containerBuilder->getResources()
18      );
19  }
20
21  require_once $file;
22  $container = new MyCachedContainer();
```

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

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

Chapter 19
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 19-1
```php
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 19-2
```yaml
parameters:
    acme_mailer.transport_chain.class: TransportChain

services:
    acme_mailer.transport_chain:
        class: "%acme_mailer.transport_chain.class%"
```

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 19-3

```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 19-4

```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->hasDefinition('acme_mailer.transport_chain')) {
            return;
        }

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

        $taggedServices = $container->findTaggedServiceIds(
            'acme_mailer.transport'
        );
        foreach ($taggedServices as $id => $attributes) {
            $definition->addMethodCall(
                'addTransport',
                array(new Reference($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 19-5
```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 TransportChain.

To begin with, change the `TransportChain` class:

Listing 19-6
```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];
        }
        else {
           return;
        }
    }
}
```

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 19-7

```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 19-8

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

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

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

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

The trickiest part is the `$attributes` variable. Because you can use the same tag many times on the same service (e.g. you could theoretically tag the same service 5 times with the `acme_mailer.transport` tag), `$attributes` is an array of the tag information for each tag on that service.

Chapter 20
Using a Factory to Create Services

Symfony2'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 object.

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

Listing 20-1
```php
class NewsletterFactory
{
    public function get()
    {
        $newsletterManager = new NewsletterManager();

        // ...

        return $newsletterManager;
    }
}
```

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

Listing 20-2
```yaml
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    newsletter_factory.class: NewsletterFactory
services:
    newsletter_manager:
        class:          "%newsletter_manager.class%"
        factory_class:  "%newsletter_factory.class%"
        factory_method: get
```

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 (as in this example), configure the factory itself as a service:

Listing 20-3

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    newsletter_factory.class: NewsletterFactory
services:
    newsletter_factory:
        class:            "%newsletter_factory.class%"
    newsletter_manager:
        class:            "%newsletter_manager.class%"
        factory_service:  newsletter_factory
        factory_method:   get
```

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.

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 `get` method in the previous example takes the `templating` service as an argument:

Listing 20-4

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    newsletter_factory.class: NewsletterFactory
services:
    newsletter_factory:
        class:            "%newsletter_factory.class%"
    newsletter_manager:
        class:            "%newsletter_manager.class%"
        factory_service:  newsletter_factory
        factory_method:   get
        arguments:
            - "@templating"
```

Chapter 21
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 a 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 21-1

```
1  class NewsletterManager 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  }
```

and also a `GreetingCardManager` class:

Listing 21-2
```
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 21-3
```
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 21-4
```
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 21-5

```
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 22

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 22-1

```
class NewsletterManager
{
    protected $mailer;
    protected $emailFormatter;

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

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }

    // ...
}
```

and also a Greeting Card class which shares the same dependencies:

Listing 22-2

```
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 22-3

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    greeting_card_manager.class: GreetingCardManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    newsletter_manager:
        class:     "%newsletter_manager.class%"
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    greeting_card_manager:
        class:     "%greeting_card_manager.class%"
        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 22-4

```
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 22-5
```php
class NewsletterManager extends MailManager
{
    // ...
}
```

and:

Listing 22-6
```php
class GreetingCardManager extends MailManager
{
    // ...
}
```

In a similar fashion, the Symfony2 service container also supports extending services in the configuration so you can also reduce the repetition by specifying a parent for a service.

Listing 22-7
```yaml
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    greeting_card_manager.class: GreetingCardManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        abstract:  true
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    newsletter_manager:
        class:     "%newsletter_manager.class%"
        parent: mail_manager

    greeting_card_manager:
        class:     "%greeting_card_manager.class%"
        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.

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 22-8

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    greeting_card_manager.class: GreetingCardManager
services:
    my_mailer:
        # ...
    my_alternative_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        abstract:  true
        calls:
            - [setMailer, ["@my_mailer"]]
            - [setEmailFormatter, ["@my_email_formatter"]]

    newsletter_manager:
        class:     "%newsletter_manager.class%"
        parent: mail_manager
        calls:
            - [setMailer, ["@my_alternative_mailer"]]

    greeting_card_manager:
        class:     "%greeting_card_manager.class%"
        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.

Collections of Dependencies

It should be noted that the overridden setter method in the previous example is actually called twice - once per the parent definition and once per the child definition. In the previous example, that was fine, since the second `setMailer` call replaces mailer object set by the first call.

In some cases, however, this can be a problem. For example, if the overridden method call involves adding something to a collection, then two objects will be added to that collection. The following shows such a case, if the parent class looks like this:

Listing 22-9

```
abstract class MailManager
{
```

```
    protected $filters;

    public function setFilter($filter)
    {
        $this->filters[] = $filter;
    }

    // ...
}
```

If you had the following config:

Listing 22-10

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
services:
    my_filter:
        # ...
    another_filter:
        # ...
    mail_manager:
        abstract:  true
        calls:
            - [setFilter, ["@my_filter"]]

    newsletter_manager:
        class:     "%newsletter_manager.class%"
        parent: mail_manager
        calls:
            - [setFilter, ["@another_filter"]]
```

In this example, the `setFilter` of the `newsletter_manager` service will be called twice, resulting in the `$filters` array containing both `my_filter` and `another_filter` objects. This is great if you just want to add additional filters to the subclasses. If you want to replace the filters passed to the subclass, removing the parent setting from the config will prevent the base class from calling `setFilter`.

In the examples shown there is a similar relationship between the parent and child services and the underlying parent and child classes. This does not need to be the case though, you can extract common parts of similar service definitions into a parent service without also inheriting a parent class.

Chapter 23
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 as you can access it via:

Listing 23-1
```
1 $doctrine = $container->get('doctrine');
```

However, there are use-cases when you don't want a service to be public. This is common when a service is only defined because it could be used as an argument for another service.

If you use a private service as an argument to more than one other service, this will result in two different instances being used as the instantiation of the private service is done inline (e.g. `new PrivateFooBar()`).

Simply said: A service will be private when you do not want to access it directly from your code.

Here is an example:

Listing 23-2
```
1 services:
2    foo:
3      class: Example\Foo
4      public: false
```

Now that the service is private, you *cannot* call:

Listing 23-3
```
1 $container->get('foo');
```

However, if a service has been marked as private, you can still alias it (see below) to access this service (via the alias).

Services are by default public.

Aliasing

You may sometimes want to use shortcuts to access some services. You can do so by aliasing them and, furthermore, you can even alias non-public services.

Listing 23-4
```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 23-5
```php
$container->get('bar'); // Would return the foo service
```

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 23-6
```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 function require_once which means that your file will be included only once per request.

Chapter 24
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 Symfony2 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 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 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 25

The Event Dispatcher Component

Introduction

Objected 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 he wants to share his 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 Symfony2 Event Dispatcher component implements the *Observer*[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 *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 Symfony2 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 Symfony2 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.

Installation

You can install the component in many different ways:

1. `http://en.wikipedia.org/wiki/Observer_pattern`

- Use the official Git repository (*https://github.com/symfony/EventDispatcher*[2]);
- *Install it via Composer* (`symfony/event-dispatcher` on *Packagist*[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 25-1
```
1 use Symfony\Component\EventDispatcher\EventDispatcher;
2
3 $dispatcher = new EventDispatcher();
```

2. https://github.com/symfony/EventDispatcher
3. https://packagist.org/packages/symfony/event-dispatcher
4. http://api.symfony.com/2.0/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html

Connecting Listeners

To take advantage of an existing event, you need to connect a listener to the dispatcher so that it can be notified when the event is dispatched. A call to the dispatcher `addListener()` method associates any valid PHP callable to an event:

Listing 25-2
```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) 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 25-3
```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 25-4
```php
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

    public function onFooAction(Event $event)
    {
        // ... do something
    }
}
```

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.event` event passes an instance of `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

6. http://www.php.net/manual/en/language.pseudo-types.php#language.types.callback
7. http://php.net/manual/en/functions.anonymous.php

Listing 25-5
```php
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    $request = $event->getRequest();

    // ...
}
```

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 25-6
```php
namespace Acme\StoreBundle;

final class StoreEvents
{
    /**
     * The store.order event is thrown each time an order is created
     * in the system.
     *
     * The event listener receives an
     * Acme\StoreBundle\Event\FilterOrderEvent instance.
     *
     * @var string
     */
    const STORE_ORDER = 'store.order';
}
```

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 25-7
```php
namespace Acme\StoreBundle\Event;

```

```php
use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Order;

class FilterOrderEvent extends Event
{
    protected $order;

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

    public function getOrder()
    {
        return $this->order;
    }
}
```

Each listener now has access to the `Order` object via the `getOrder` method.

Dispatch the Event

The *`dispatch()`*[8] 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 25-8

```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 25-9

```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
}
```

Passing along the Event Dispatcher Object

If you have a look at the `EventDispatcher` class, you will notice that the class does not act as a Singleton (there is no `getInstance()` static method). That is intentional, as you might want to have several

8. http://api.symfony.com/2.0/Symfony/Component/EventDispatcher/EventDispatcher.html#dispatch()

concurrent event dispatchers in a single PHP request. But it also means that you need a way to pass the dispatcher to the objects that need to connect or notify events.

The best practice is to inject the event dispatcher object into your objects, aka dependency injection.

You can use constructor injection:

Listing 25-10
```php
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class Foo
{
    protected $dispatcher = null;

    public function __construct(EventDispatcherInterface $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Or setter injection:

Listing 25-11
```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.

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`*[9] 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 25-12
```php
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class StoreSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
```

9. http://api.symfony.com/2.0/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

```
    {
        return array(
            'kernel.response' => array(
                array('onKernelResponsePre', 10),
                array('onKernelResponseMid', 5),
                array('onKernelResponsePost', 0),
            ),
            'store.order'     => array('onStoreOrder', 0),
        );
    }

    public function onKernelResponsePre(FilterResponseEvent $event)
    {
        // ...
    }

    public function onKernelResponseMid(FilterResponseEvent $event)
    {
        // ...
    }

    public function onKernelResponsePost(FilterResponseEvent $event)
    {
        // ...
    }

    public function onStoreOrder(FilterOrderEvent $event)
    {
        // ...
    }
}
```

This is very similar to a listener class, except that the class itself can tell the dispatcher which events it should listen to. To register a subscriber with the dispatcher, use the *addSubscriber()*[10] method:

Listing 25-13
```
use Acme\StoreBundle\Event\StoreSubscriber;

$subscriber = new StoreSubscriber();
$dispatcher->addSubscriber($subscriber);
```

The dispatcher will automatically register the subscriber for each event returned by the `getSubscribedEvents` method. This method returns an array indexed by event names and whose values are either the method name to call or an array composed of the method name to call and a priority. The example above shows how to register several listener methods for the same event in subscriber 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()*[11] method:

10. http://api.symfony.com/2.0/Symfony/Component/EventDispatcher/EventDispatcher.html#addSubscriber()
11. http://api.symfony.com/2.0/Symfony/Component/EventDispatcher/Event.html#stopPropagation()

Listing 25-14
```php
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    // ...

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

Now, any listeners to `store.order` that have not yet been called will *not* be called.

Chapter 26
The Finder Component

The Finder Component finds files and directories via an intuitive fluent interface.

Installation

You can install the component in many different ways:

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

Usage

The *`Finder`*[3] class finds files and/or directories:

Listing 26-1
```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
    print $file->getRelativePathname()."\n";
}
```

1. https://github.com/symfony/Finder
2. https://packagist.org/packages/symfony/finder
3. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html

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 26-2
```
1 $finder->in(__DIR__);
```

Search in several locations by chaining calls to *in()*[12]:

Listing 26-3
```
1 $finder->files()->in(__DIR__)->in('/elsewhere');
```

Exclude directories from matching with the *exclude()*[13] method:

Listing 26-4
```
1 $finder->in(__DIR__)->exclude('ruby');
```

As the Finder uses PHP iterators, you can pass any URL with a supported *protocol*[14]:

Listing 26-5
```
1 $finder->in('ftp://example.com/pub/');
```

And it also works with user-defined streams:

Listing 26-6

4. http://api.symfony.com/2.0/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/2.0/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/2.0/Symfony/Component/Finder/Finder.html#in()
13. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#exclude()
14. http://www.php.net/manual/en/wrappers.php

```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*[15] documentation to learn how to create your own streams.

Files or Directories

By default, the Finder returns files and directories; but the *`files()`*[16] and *`directories()`*[17] methods control that:

Listing 26-7
```php
$finder->files();

$finder->directories();
```

If you want to follow links, use the `followLinks()` method:

Listing 26-8
```php
$finder->files()->followLinks();
```

By default, the iterator ignores popular VCS files. This can be changed with the `ignoreVCS()` method:

Listing 26-9
```php
$finder->ignoreVCS(false);
```

Sorting

Sort the result by name or by type (directories first, then files):

Listing 26-10
```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:

15. http://www.php.net/streams
16. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#files()
17. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#directories()

Listing 26-11
```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()*[18] method:

Listing 26-12
```php
$finder->files()->name('*.php');
```

The `name()` method accepts globs, strings, or regexes:

Listing 26-13
```php
$finder->files()->name('/\.php$/');
```

The `notName()` method excludes files matching a pattern:

Listing 26-14
```php
$finder->files()->notName('*.rb');
```

File Size

Restrict files by size with the *size()*[19] method:

Listing 26-15
```php
$finder->files()->size('< 1.5K');
```

Restrict by a size range by chaining calls:

Listing 26-16
```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*[20].

File Date

Restrict files by last modified dates with the *date()*[21] method:

Listing 26-17
```php
$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*[22] function.

---

18. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#name()
19. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#size()
20. http://physics.nist.gov/cuu/Units/binary.html
21. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#date()
22. http://www.php.net/manual/en/datetime.formats.php

Directory Depth

By default, the Finder recursively traverse directories. Restrict the depth of traversing with *depth()*[23]:

Listing 26-18
```
$finder->depth('== 0');
$finder->depth('< 3');
```

Custom Filtering

To restrict the matching file with your own strategy, use *filter()*[24]:

Listing 26-19
```
$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*[25] instance. The file is excluded from the result set if the Closure returns `false`.

23. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#depth()
24. http://api.symfony.com/2.0/Symfony/Component/Finder/Finder.html#filter()
25. http://api.symfony.com/2.0/Symfony/Component/Finder/SplFileInfo.html

Chapter 27
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 Symfony2 HttpFoundation component replaces these default PHP global variables and functions by an Object-Oriented layer.

Installation

You can install the component in many different ways:

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

Request

The most common way to create request is to base it on the current PHP global variables with *`createFromGlobals()`*[3]:

Listing 27-1
```php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

which is almost equivalent to the more verbose, but also more flexible, *`__construct()`*[4] call:

Listing 27-2

1. https://github.com/symfony/HttpFoundation
2. https://packagist.org/packages/symfony/http-foundation
3. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#createFromGlobals()
4. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#__construct()

```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('Content-Type')`).

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;
- *add()*[17]: Adds parameters;
- *get()*[18]: Returns a parameter by name;
- *set()*[19]: Sets a parameter by name;

---

5. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
6. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
7. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
8. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
9. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
10. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/FileBag.html
11. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ServerBag.html
12. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/HeaderBag.html
13. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
14. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#all()
15. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#keys()
16. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#replace()
17. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#add()
18. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#get()
19. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#set()

- *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()` 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 27-3
```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'
```

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()*[28] via the third argument:

Listing 27-4
```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'
```

Finally, you can also store additional data in the request, thanks to the public `attributes` property, which is also an instance of *ParameterBag*[29]. This is mostly used to attach information that belongs to the Request and that needs to be accessed from many different points in your application. For information on how this is used in the Symfony2 framework, see *read more*.

20. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#has()
21. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#remove()
22. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
23. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#getAlpha()
24. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#getAlnum()
25. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#getDigits()
26. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#getInt()
27. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#filter()
28. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#get()
29. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html

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()*[30] method:

Listing 27-5
```
// 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 27-6
```
$request = Request::create(
    '/hello-world',
    'GET',
    array('name' => 'Fabien')
);
```

The *create()*[31] method creates a request based on a path info, a method and some parameters (the query parameters or the request ones depending on the HTTP method); and of course, you can also override all other variables as well (by default, Symfony creates sensible defaults for all the PHP global variables).

Based on such a request, you can override the PHP global variables via *overrideGlobals()*[32]:

Listing 27-7
```
$request->overrideGlobals();
```

You can also duplicate an existing query via *duplicate()*[33] or change a bunch of parameters with a single call to *initialize()*[34].

Accessing the Session

If you have a session attached to the Request, you can access it via the *getSession()*[35] method; the *hasPreviousSession()*[36] 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()*[37]: returns the list of accepted content types ordered by descending quality;
- *getLanguages()*[38]: returns the list of accepted languages ordered by descending quality;
- *getCharsets()*[39]: returns the list of accepted charsets ordered by descending quality;

30. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getPathInfo()
31. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#create()
32. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#overrideGlobals()
33. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#duplicate()
34. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#initialize()
35. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getSession()
36. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#hasPreviousSession()
37. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getAcceptableContentTypes()
38. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getLanguages()

Accessing other Data

The Request class has many other methods that you can use to access the request information. Have a look at the API for more information about them.

Response

A *Response*[40] 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 27-8
```php
use Symfony\Component\HttpFoundation\Response;

$response = new Response(
    'Content',
    200,
    array('content-type' => 'text/html')
);
```

These information can also be manipulated after the Response object creation:

Listing 27-9
```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()*[41] method:

Listing 27-10
```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()*[42] method:

Listing 27-11
```php
$response->prepare($request);
```

Sending the response to the client is then as simple as calling *send()*[43]:

Listing 27-12
```php
$response->send();
```

39. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getCharsets()
40. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html
41. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setCharset()
42. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#prepare()
43. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#send()

Setting Cookies

The response cookies can be manipulated though the `headers` public attribute:

Listing 27-13
```php
use Symfony\Component\HttpFoundation\Cookie;

$response->headers->setCookie(new Cookie('foo', 'bar'));
```

The *setCookie()*[44] method takes an instance of *Cookie*[45] as an argument.

You can clear a cookie via the *clearCookie()*[46] method.

Managing the HTTP Cache

The *Response*[47] class has a rich set of methods to manipulate the HTTP headers related to the cache:

- *setPublic()*[48];
- *setPrivate()*[49];
- *expire()*[50];
- *setExpires()*[51];
- *setMaxAge()*[52];
- *setSharedMaxAge()*[53];
- *setTtl()*[54];
- *setClientTtl()*[55];
- *setLastModified()*[56];
- *setEtag()*[57];
- *setVary()*[58];

The *setCache()*[59] method can be used to set the most commonly used cache information in one method call:

Listing 27-14
```php
$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()*[60] method:

---

44. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#setCookie()
45. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Cookie.html
46. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#clearCookie()
47. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html
48. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setPublic()
49. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setPrivate()
50. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#expire()
51. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setExpires()
52. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setMaxAge()
53. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setSharedMaxAge()
54. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setTtl()
55. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setClientTtl()
56. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setLastModified()
57. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setEtag()
58. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setVary()
59. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setCache()

Listing 27-15
```php
if ($response->isNotModified($request)) {
    $response->send();
}
```

If the Response is not modified, it sets the status code to 304 and remove the actual response content.

Redirecting the User

To redirect the client to another URL, you can use the *RedirectResponse*[61] class:

Listing 27-16
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://example.com/');
```

Creating a JSON Response

Any type of response can be created via the *Response*[62] class by setting the right content and headers. A JSON response might look like this:

Listing 27-17
```php
use Symfony\Component\HttpFoundation\Response;

$response = new Response();
$response->setContent(json_encode(array(
    'data' => 123,
)));
$response->headers->set('Content-Type', 'application/json');
```

Session

TBD -- This part has not been written yet as it will probably be refactored soon in Symfony 2.1.

60. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#isNotModified()
61. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/RedirectResponse.html
62. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html

Chapter 28
Trusting Proxies

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, Symfony2 does *not* trust these proxy headers by default. If you are behind a proxy, you should manually whitelist your proxy:

Listing 28-1
```php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
// only trust proxy headers coming from this IP address
$request->setTrustedProxies(array('192.0.0.1'));
```

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];

If your reverse proxy uses a different header name for any of these, you can configure that header name via *`setTrustedHeaderName()`*[6]:

Listing 28-2

1. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getClientIp()
2. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getHost()
3. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getPort()
4. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getScheme()
5. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#isSecure()
6. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#setTrustedHeaderName()

```
$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 28-3
```
// disables trusting the ``X-Forwarded-Proto`` header, the default header is used
$request->setTrustedHeaderName(Request::HEADER_CLIENT_PROTO, '');
```

Chapter 29
The HttpKernel Component

The HttpKernel Component provides a structured process for converting a `Request` into a `Response` by making use of the event dispatcher. 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 many different ways:

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

../../_images/request-response-flow.png

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 29-1
```php
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;
```

1. https://github.com/symfony/HttpKernel
2. https://packagist.org/packages/symfony/http-kernel

```
4
5  interface HttpKernelInterface
6  {
7      // ...
8
9      /**
10      * @return Response A Response instance
11      */
12     public function handle(
13         Request $request,
14         $type = self::MASTER_REQUEST,
15         $catch = true
16     );
17 }
```

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].

../../_images/01-workflow.png

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 *event dispatcher* and a *controller resolver* (explained below). To complete your working kernel, you'll add more event listeners to the events discussed below:

Listing 29-2
```
1  use Symfony\Component\HttpFoundation\Request;
2  use Symfony\Component\HttpKernel\HttpKernel;
3  use Symfony\Component\EventDispatcher\EventDispatcher;
4  use Symfony\Component\HttpKernel\Controller\ControllerResolver;
5
6  // create the Request object
7  $request = Request::createFromGlobals();
8
9  $dispatcher = new EventDispatcher();
10 // ... add some event listeners
11
12 // create your controller resolver
13 $resolver = new ControllerResolver();
14 // instantiate the kernel
15 $kernel = new HttpKernel($dispatcher, $resolver);
```

3. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html#handle()
4. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernelInterface.html#handle()
5. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
6. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html
7. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html

```
16
17 // actually execute the kernel, which turns the request into a response
18 // by dispatching events, calling a controller, and returning the response
19 $response = $kernel->handle($request);
20
21 // echo the content and send the headers
22 $response->send();
```

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 Symfony2 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.

../../_images/02-kernel-request.png

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.

../../_images/03-kernel-request-response.png

Other listeners simply initialize things or add more information to the request. For example, a listener might determine and set the locale on the Session 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).

8. http://fabien.potencier.org/article/50/create-your-own-framework-on-top-of-the-symfony2-components-part-1
9. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html#handle()
10. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/RedirectResponse.html

`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 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`.

../../_images/04-resolve-controller.png

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 29-3
```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.

11. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/RouterListener.html
12. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
13. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html
14. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
15. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getController()
16. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()

Resolving the Controller in the Symfony2 Framework

The Symfony Framework uses the built-in *`ControllerResolver`*[17] class (actually, it uses a sub-class, which 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:

a) 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 Symfony2 - e.g. `Acme\DemoBundle\Controller\DefaultController::indexAction`. This transformation is specific to the *`ControllerResolver`*[18] sub-class used by the Symfony2 Framework.

b) A new instance of your controller class is instantiated with no constructor arguments.

c) 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 Symfony2 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 Symfony2 section below.

../../_images/06-kernel-controller.png

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.

17. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
18. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html
19. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/ContainerAwareInterface.html
20. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.html
21. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html#setController()

`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*, which is packaged with the Symfony Standard Edition. This listener's *@ParamConverter* functionality allows you to pass a full object (e.g. a `Post` object) to your controller instead of a scalar value (e.g. an `id` parameter that was on your route). The listener - `ParamConverterListener` - uses reflection to look at each of the arguments of the controller and tries to use different methods to convert those to objects, which are then stored in the `attributes` property of the `Request` object. Read the next section to see why this is important.

4) Getting the Controller Arguments

Next, `HttpKernel::handle` calls *`getArguments()`*[22]. Remember that the controller returned in `getController` is a callable. The purpose of `getArguments` is to return the array of arguments that should be passed to that controller. Exactly how this is done is completely up to your design, though the built-in *`ControllerResolver`*[23] is a good example.

../../_images/07-controller-arguments.png

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 Symfony2 Framework

Now that you know exactly what the controller callable (usually a method inside a controller object) is, the `ControllerResolver` uses *reflection*[24] on the callable to return an array of the *names* of each of the arguments. It then iterates over each of these arguments and uses the following tricks to determine which value should be passed for each argument:

a) 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`).

b) If the argument in the controller is type-hinted with Symfony's *`Request`*[25] object, then the `Request` is passed in as the value.

5) Calling the Controller

The next step is simple! `HttpKernel::handle` executes the controller.

../../_images/08-call-controller.png

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.

22. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()
23. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
24. http://php.net/manual/en/book.reflection.php
25. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html

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.

../../_images/09-controller-returns-response.png

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`.

../../_images/10-kernel-view.png

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`.

`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 - does* add a listener to this event. If your controller returns an array, and you place the *@Template* 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*[26] implements a listener on this event which aims to give you a robust view layer capable of using a single controller to return many different content-type responses (e.g. HTML, JSON, XML, etc).

7) The `kernel.response` event

Typical Purposes: Modify the `Response` object just before it is sent

Kernel Events Information Table

The end goal of the kernel is to transform a `Request` into a `Response`. The `Response` might be created during the *kernel.request* event, returned from the *controller*, or returned by one of the listeners to the *kernel.view* event.

Regardless of who creates the `Response`, another event - `kernel.response` is dispatched directly afterwards. A typical listener to this event will modify the `Response` object in some way, such as

26. `https://github.com/friendsofsymfony/FOSRestBundle`

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()`*[27]. In the most typical use-case, you can then call the *`send()`*[28] 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`*[29] 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`*[30] serializes the current user's information into the session so that it can be reloaded on the next 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.

../../_images/11-kernel-exception.png

Each listener to this event is passed a *`GetResponseForExceptionEvent`*[31] object, which you can use to access the original exception via the *`getException()`*[32] method. A typical listener on this event will check for a certain type of exception and create an appropriate error `Response`.

For example, to generate a 404 page, you might throw a special type of exception and then add a listener on this event that looks for this exception and creates and returns a 404 `Response`. In fact, the `HttpKernel` component comes with an *`ExceptionListener`*[33], which if you choose to use, will do this and more by default (see the sidebar below for more details).

27. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html#handle()
28. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#send()
29. http://api.symfony.com/2.0/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
30. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/ContextListener.html
31. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
32. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html#getException()
33. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html

`kernel.exception` in the Symfony Framework

There are two main listeners to `kernel.exception` when using the Symfony Framework.

ExceptionListener in HttpKernel

The first comes core to the `HttpKernel` component and is called *`ExceptionListener`*[34]. The listener has several goals:

1) The thrown exception is converted into a *`FlattenException`*[35] object, which contains all the information about the request, but which can be printed and serialized.

2) If the original exception implements *`HttpExceptionInterface`*[36], 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`*[37]. 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 Event Dispatcher Component*.

The name of each of the "kernel" events is defined as a constant on the *`KernelEvents`*[38] class. Additionally, each event listener is passed a single argument, which is some sub-class of *`KernelEvent`*[39]. 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`*[40]
kernel.controller	`KernelEvents::CONTROLLER`	*`FilterControllerEvent`*[41]
kernel.view	`KernelEvents::VIEW`	*`GetResponseForControllerResultEvent`*[42]
kernel.response	`KernelEvents::RESPONSE`	*`FilterResponseEvent`*[43]
kernel.exception	`KernelEvents::EXCEPTION`	*`GetResponseForExceptionEvent`*[44]

34. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html
35. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Exception/FlattenException.html
36. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.html
37. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/ExceptionListener.html
38. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/KernelEvents.html
39. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
40. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html
41. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html
42. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
43. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html
44. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html

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`*[45]. 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 29-4

```
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\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();
```

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).

../../_images/sub-request.png

To execute a sub request, use `HttpKernel::handle`, but change the second arguments as follows:

Listing 29-5

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernelInterface;

// ...
```

45. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html

```php

// create some other request manually as needed
$request = new Request();
// for example, possibly set its _controller manually
$request->attributes->add('_controller', '...');

$response = $kernel->handle($request, HttpKernelInterface::SUB_REQUEST);
// do something with this response
```

This creates another full request-response cycle where this new `Request` is transformed into a `Response`. The only difference internally is that some listeners (e.g. security) may only act upon the master request. Each listener is passed some sub-class of *`KernelEvent`*[46], whose *`getRequestType()`*[47] 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:

Listing 29-6

```php
use Symfony\Component\HttpKernel\HttpKernelInterface;
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
        return;
    }

    // ...
}
```

46. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
47. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/KernelEvent.html#getRequestType()

Chapter 30
The Locale Component

> Locale component provides fallback code to handle cases when the `intl` extension is missing. Additionally it extends the implementation of a native *`Locale`*[1] class with several handy methods.

Replacement for the following functions and classes is provided:

- *`intl_is_failure`*[2]
- *`intl_get_error_code`*[3]
- *`intl_get_error_message`*[4]
- *`Collator`*[5]
- *`IntlDateFormatter`*[6]
- *`Locale`*[7]
- *`NumberFormatter`*[8]

Stub implementation only supports the `en` locale.

Installation

You can install the component in many different ways:

- Use the official Git repository (*https://github.com/symfony/Locale*[9]);

1. http://php.net/manual/en/class.locale.php
2. http://php.net/manual/en/function.intl-is-failure.php
3. http://php.net/manual/en/function.intl-get-error-code.php
4. http://php.net/manual/en/function.intl-get-error-message.php
5. http://php.net/manual/en/class.collator.php
6. http://php.net/manual/en/class.intldateformatter.php
7. http://php.net/manual/en/class.locale.php
8. http://php.net/manual/en/class.numberformatter.php
9. https://github.com/symfony/Locale

- *Install it via Composer* (`symfony/locale` on *Packagist*[10]).

Usage

Taking advantage of the fallback code includes requiring function stubs and adding class stubs to the autoloader.

When using the ClassLoader component following code is sufficient to supplement missing `intl` extension:

Listing 30-1
```php
if (!function_exists('intl_get_error_code')) {
    require __DIR__.'/path/to/src/Symfony/Component/Locale/Resources/stubs/functions.php';

    $loader->registerPrefixFallbacks(
        array(__DIR__.'/path/to/src/Symfony/Component/Locale/Resources/stubs')
    );
}
```

Locale[11] class enriches native *Locale*[12] class with additional features:

Listing 30-2
```php
use Symfony\Component\Locale\Locale;

// Get the country names for a locale or get all country codes
$countries = Locale::getDisplayCountries('pl');
$countryCodes = Locale::getCountries();

// Get the language names for a locale or get all language codes
$languages = Locale::getDisplayLanguages('fr');
$languageCodes = Locale::getLanguages();

// Get the locale names for a given code or get all locale codes
$locales = Locale::getDisplayLocales('en');
$localeCodes = Locale::getLocales();
```

10. https://packagist.org/packages/symfony/locale
11. http://api.symfony.com/2.0/Symfony/Component/Locale/Locale.html
12. http://php.net/manual/en/class.locale.php

Chapter 31
The Process Component

The Process Component executes commands in sub-processes.

Installation

You can install the component in many different ways:

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

Usage

The *`Process`*[3] class allows you to execute a command in a sub-process:

Listing 31-1
```php
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->setTimeout(3600);
$process->run();
if (!$process->isSuccessful()) {
    throw new \RuntimeException($process->getErrorOutput());
}

print $process->getOutput();
```

The *`run()`*[4] method takes care of the subtle differences between the different platforms when executing the command.

1. https://github.com/symfony/Process
2. https://packagist.org/packages/symfony/process
3. http://api.symfony.com/2.0/Symfony/Component/Process/Process.html
4. http://api.symfony.com/2.0/Symfony/Component/Process/Process.html#run()

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()*[5] method:

Listing 31-2
```php
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->run(function ($type, $buffer) {
    if ('err' === $type) {
        echo 'ERR > '.$buffer;
    } else {
        echo 'OUT > '.$buffer;
    }
});
```

If you want to execute some PHP code in isolation, use the `PhpProcess` instead:

Listing 31-3
```php
use Symfony\Component\Process\PhpProcess;

$process = new PhpProcess(<<<EOF
    <?php echo 'Hello World'; ?>
EOF
);
$process->run();
```

5. http://api.symfony.com/2.0/Symfony/Component/Process/Process.html#run()

Chapter 32
The Routing Component

The Routing Component maps an HTTP request to a set of configuration variables.

Installation

You can install the component in many different ways:

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

Let's see a quick example. Notice that this assumes that you've already configured your autoloader to load the Routing component:

Listing 32-1
```
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;
5
```

1. https://github.com/symfony/Routing
2. https://packagist.org/packages/symfony/routing
3. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html
4. http://api.symfony.com/2.0/Symfony/Component/Routing/Route.html
5. http://api.symfony.com/2.0/Symfony/Component/Routing/RequestContext.html
6. http://api.symfony.com/2.0/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 four parts:

1. The URL pattern 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.

Take the following route, which combines several of these ideas:

Listing 32-2

```php
$route = new Route(
    '/archive/{month}', // path
    array('controller' => 'showArchive'), // default values
    array('month' => '[0-9]{4}-[0-9]{2}'), // requirements
    array() // options
);

// ...

$parameters = $matcher->match('/archive/2012-01');
```

7. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html
8. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html#add()
9. http://api.symfony.com/2.0/Symfony/Component/Routing/Route.html
10. http://api.symfony.com/2.0/Symfony/Component/Routing/Exception/ResourceNotFoundException.html

```
// array(
//     'controller' => 'showArchive',
//     'month'      => '2012-01',
//     '_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.

Besides the regular expression constraints there are two special requirements you can define:

- `_method` enforces a certain HTTP request method (`HEAD`, `GET`, `POST`, ...)
- `_scheme` enforces a certain HTTP scheme (`http`, `https`)

For example, the following route would only accept requests to /foo with the POST method and a secure connection:

Listing 32-3
```
$route = new Route(
    '/foo',
    array(),
    array('_method' => 'post', '_scheme' => 'https' )
);
```

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 32-4
```
$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, default requirements and default options to all routes of a subtree:

Listing 32-5
```
$rootCollection = new RouteCollection();

$subCollection = new RouteCollection();
$subCollection->add(...);
$subCollection->add(...);

$rootCollection->addCollection(
    $subCollection,
    '/prefix',
```

11. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html

```
10      array('_scheme' => 'https')
11  );
```

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 32-6

```
1  public function __construct(
2      $baseUrl = '',
3      $method = 'GET',
4      $host = 'localhost',
5      $scheme = 'http',
6      $httpPort = 80,
7      $httpsPort = 443
8  )
```

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 32-7

```
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 32-8

```
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/2.0/Symfony/Component/Routing/RequestContext.html
13. http://api.symfony.com/2.0/Symfony/Component/Routing/RequestContext.html
14. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
15. http://api.symfony.com/2.0/Symfony/Component/Routing/RequestContext.html
16. http://api.symfony.com/2.0/Symfony/Component/Routing/Matcher/UrlMatcher.html

If you have defined the `_scheme` requirement, 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 32-9
```yaml
# routes.yml
route1:
    pattern: /foo
    defaults: { _controller: 'MyController::fooAction' }

route2:
    pattern: /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 32-10
```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 32-11
```php
// RouteProvider.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
```

17. http://api.symfony.com/2.0/Symfony/Component/Routing/RequestContext.html
18. http://api.symfony.com/2.0/Symfony/Component/Config/FileLocator.html
19. http://api.symfony.com/2.0/Symfony/Component/Config/FileLocator.html
20. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html
21. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/YamlFileLoader.html
22. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/XmlFileLoader.html
23. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/PhpFileLoader.html
24. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/PhpFileLoader.html
25. http://api.symfony.com/2.0/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 32-12
```
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 a 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 32-13
```
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 32-14
```
1  $locator = new FileLocator(array(__DIR__));
2  $requestContext = new RequestContext($_SERVER['REQUEST_URI']);
3
```

26. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/ClosureLoader.html
27. http://api.symfony.com/2.0/Symfony/Component/Routing/RouteCollection.html
28. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/AnnotationDirectoryLoader.html
29. http://api.symfony.com/2.0/Symfony/Component/Routing/Loader/AnnotationFileLoader.html
30. http://api.symfony.com/2.0/Symfony/Component/Routing/Router.html
31. http://api.symfony.com/2.0/Symfony/Component/Routing/Router.html

```php
$router = new Router(
    new YamlFileLoader($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 33
The Security Component

Introduction

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 many different ways:

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

Sections

- *The Firewall and Security Context*
- *Authentication*
- *Authorization*

1. `https://github.com/symfony/Security`
2. `https://packagist.org/packages/symfony/security`

Chapter 34

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 34-1
```php
use Symfony\Component\Security\SecurityContext;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

$securityContext = new SecurityContext();

// ... authenticate the user

if (!$securityContext->isGranted('ROLE_ADMIN')) {
    throw new AccessDeniedException();
}
```

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 34-2
```php
use Symfony\Component\Security\Http\FirewallMap;
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\ExceptionListener;

$map = new FirewallMap();

```

1. http://api.symfony.com/2.0/Symfony/Component/Security/Core/SecurityContextInterface.html

```php
$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 34-3
```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`).

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 his credentials again (when he has 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

2. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html
3. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/ListenerInterface.html
4. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html
5. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Exception/AccessDeniedHttpException.html

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 his 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*.

6. http://api.symfony.com/2.0/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html
7. http://api.symfony.com/2.0/Symfony/Component/Security/Http/EntryPoint/AuthenticationEntryPointInterface.html#start()
8. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
9. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html

Chapter 35
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 35-1
```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/2.0/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 35-2

```
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/2.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html
3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/AuthenticationProviderManager.html
4. http://api.symfony.com/2.0/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 he 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 35-3

```php
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/2.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html
6. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html#supports()
7. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface::authenticate.html
8. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html
9. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
10. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
11. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
28
29 $provider->authenticate($unauthenticatedToken);
```

The example above demonstrates the use of the "in-memory" user provider, but you may use any user provider, as long as it implements *UserProviderInterface*[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 35-4

```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.

Password Encoders

When the *getEncoder()*[17] method of the password encoder factory is called with the user object as its first argument, it will return an encoder of type *PasswordEncoderInterface*[18] which should be used to encode this user's password:

Listing 35-5

```php
// fetch a user of type Acme\Entity\LegacyUser
$user = ...

$encoder = $encoderFactory->getEncoder($user);

// will return $weakEncoder (see above)

$encodedPassword = $encoder->encodePassword($password, $user->getSalt());

```

12. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
13. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/ChainUserProvider.html
14. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Provider/DaoAuthenticationProvider.html
15. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Encoder/EncoderFactory.html
16. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html
17. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Encoder/EncoderFactory.html#getEncoder()
18. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Encoder/PasswordEncoderInterface.html

```
10 // check if the password is valid:
11
12 $validPassword = $encoder->isPasswordValid(
13     $user->getPassword(),
14     $password,
15     $user->getSalt());
```

Chapter 36
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 on which 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/2.0/Symfony/Component/Security/Core/SecurityContextInterface.html
2. http://api.symfony.com/2.0/Symfony/Component/Security/Core/SecurityContextInterface.html#setToken()
3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/AccessDecisionManagerInterface.html
4. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#getRoles()
5. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/AccessDecisionManager.html

verdict based on all the votes (either positive, negative or neutral) it has received. It recognizes several strategies:

- **`affirmative` (default)**

  grant access as soon as 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 36-1

```
use Symfony\Component\Security\Core\Authorization\AccessDecisionManager;

// instances of Symfony\Component\Security\Core\Authorization\Voter\VoterInterface
$voters = array(...);

// one of "affirmative", "consensus", "unanimous"
$strategy = ...;

// whether or not to grant access when all voters abstain
$allowIfAllAbstainDecisions = ...;

// whether or not to grant access when there is no majority (applies only to the
"consensus" strategy)
$allowIfEqualGrantedDeniedDecisions = ...;

$accessDecisionManager = new AccessDecisionManager(
    $voters,
    $strategy,
    $allowIfAllAbstainDecisions,
    $allowIfEqualGrantedDeniedDecisions
);
```

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`;

6. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html
7. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html

The security component contains some standard voters which cover many use cases:

AuthenticatedVoter

The *AuthenticatedVoter*[8] voter supports the attributes `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY` and grants access based on the current level of authentication, i.e. is the user fully authenticated, or only based on a "remember-me" cookie, or even authenticated anonymously?

Listing 36-2
```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 36-3
```php
use Symfony\Component\Security\Core\Authorization\Voter\RoleVoter;

$roleVoter = new RoleVoter('ROLE_');

$roleVoter->vote($token, $object, '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 36-4
```php
use Symfony\Component\Security\Core\Authorization\Voter\RoleHierarchyVoter;
use Symfony\Component\Security\Core\Role\RoleHierarchy;

$hierarchy = array(
```

8. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/AuthenticatedVoter.html
9. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html
10. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html#getRoles()
11. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/RoleHierarchyVoter.html
12. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/RoleVoter.html

```
    'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_USER'),
);

$roleHierarchy = new RoleHierarchy($hierarchy);

$roleHierarchyVoter = new RoleHierarchyVoter($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 36-5

```
use Symfony\Component\Security\Core\Role\Role;

$role = new Role('ROLE_ADMIN');

// will echo 'ROLE_ADMIN'
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 36-6

13. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Role/RoleInterface.html
14. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Role/Role/RoleInterface.html#getRole()
15. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Role/Role.html
16. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Token/AbstractToken.html
17. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/AccessListener.html
18. http://api.symfony.com/2.0/Symfony/Component/Security/Http/AccessMapInterface.html

```php
use Symfony\Component\Security\Http\AccessMap;
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\AccessListener;

$accessMap = new AccessMap();
$requestMatcher = new RequestMatcher('^/admin');
$accessMap->add($requestMatcher, array('ROLE_ADMIN'));

$accessListener = new AccessListener(
    $securityContext,
    $accessDecisionManager,
    $accessMap,
    $authenticationManager
);
```

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 36-7

```php
use Symfony\Component\Security\SecurityContext;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

$securityContext = new SecurityContext(
    $authenticationManager,
    $accessDecisionManager
);

if (!$securityContext->isGranted('ROLE_ADMIN')) {
    throw new AccessDeniedException();
}
```

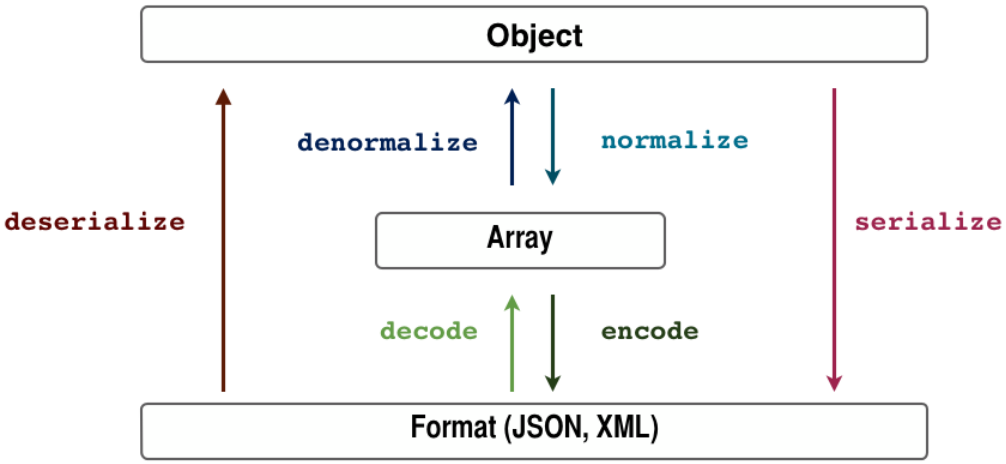
19. http://api.symfony.com/2.0/Symfony/Component/Security/Core/SecurityContext.html#isGranted()
20. http://api.symfony.com/2.0/Symfony/Component/Security/Core/SecurityContext.html

Chapter 37

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 many different ways:

- Use the official Git repository (*https://github.com/symfony/Serializer*[1]);
- *Install it via Composer* (`symfony/serializer` on *Packagist*[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 37-1
```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('xml' => new XmlEncoder(), 'json' => 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 37-2
```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;
    }
}
```

Now, if you want to serialize this object into JSON, you only need to use the Serializer service created before:

Listing 37-3

1. https://github.com/symfony/Serializer
2. https://packagist.org/packages/symfony/serializer
3. http://api.symfony.com/2.0/Symfony/Component/Serializer/Serializer.html

```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].

Deserializing an Object

Let's see now how to do the exactly the opposite. This time, the information of the *People* class would be encoded in XML format:

Listing 37-4
```php
$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
</person>
EOF;

$person = $serializer->deserialize($data,'Acme\Person','xml');
```

In this case, *deserialize()*[6] needs three parameters:

1. The information to be decoded
2. The name of the class this information will be decoded to
3. The encoder used to convert that information into an array

JMSSerializer

A popular third-party library, *JMS serializer*[7], provides a more sophisticated albeit more complex solution. This library includes the ability to configure how your objects should be serialize/deserialized via annotations (as well as YML, XML and PHP), integration with the Doctrine ORM, and handling of other complex cases (e.g. circular references).

4. http://api.symfony.com/2.0/Symfony/Component/Serializer/Serializer.html#serialize()
5. http://api.symfony.com/2.0/Symfony/Component/Serializer/Encoder/JsonEncoder.html
6. http://api.symfony.com/2.0/Symfony/Component/Serializer/Serializer.html#deserialize()
7. https://github.com/schmittjoh/serializer

Chapter 38
The Templating Component

Templating 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 many different ways:

- Use the official Git repository (*https://github.com/symfony/Templating*[1]);
- *Install it via Composer* (`symfony/templating` on *Packagist*[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 and template loader (*`LoaderInterface`*[5]) to find the template associated to a reference:

Listing 38-1
```php
use Symfony\Component\Templating\PhpEngine;
use Symfony\Component\Templating\TemplateNameParser;
use Symfony\Component\Templating\Loader\FilesystemLoader;

$loader = new FilesystemLoader(__DIR__ . '/views/%name%');

$view = new PhpEngine(new TemplateNameParser(), $loader);
```

1. https://github.com/symfony/Templating
2. https://packagist.org/packages/symfony/templating
3. http://api.symfony.com/2.0/Symfony/Component/Templating/PhpEngine.html
4. http://api.symfony.com/2.0/Symfony/Component/Templating/TemplateNameParserInterface.html
5. http://api.symfony.com/2.0/Symfony/Component/Templating/Loader/LoaderInterface.html

```
8
9  echo $view->render('hello.php', array('firstname' => 'Fabien'));
```

The *render()*[6] method executes the file *views/hello.php* and returns the output text.

Listing 38-2
```
1  <!-- views/hello.php -->
2  Hello, <?php echo $firstname ?>!
```

Template Inheritance with Slots

The template inheritance is designed to share layouts with many templates.

Listing 38-3
```
1  <!-- views/layout.php -->
2  <html>
3      <head>
4          <title><?php $view['slots']->output('title', 'Default title') ?></title>
5      </head>
6      <body>
7          <?php $view['slots']->output('_content') ?>
8      </body>
9  </html>
```

The *extend()*[7] method is called in the sub-template to set its parent template.

Listing 38-4
```
1  <!-- views/page.php -->
2  <?php $view->extend('layout.php') ?>
3
4  <?php $view['slots']->set('title', $page->title) ?>
5
6  <h1>
7      <?php echo $page->title ?>
8  </h1>
9  <p>
10     <?php echo $page->body ?>
11 </p>
```

To use template inheritance, the *SlotsHelper*[8] helper must be registered:

Listing 38-5
```
1  use Symfony\Component\Templating\Helper\SlotsHelper;
2
3  $view->set(new SlotsHelper());
4
5  // Retrieve page object
6  $page = ...;
7
8  echo $view->render('page.php', array('page' => $page));
```

6. http://api.symfony.com/2.0/Symfony/Component/Templating/PhpEngine.html#render()
7. http://api.symfony.com/2.0/Symfony/Component/Templating/PhpEngine.html#extend()
8. http://api.symfony.com/2.0/Symfony/Component/Templating/Helper/SlotsHelper.html

Multiple levels of inheritance is possible: a layout can extend an other layout.

Output Escaping

This documentation is still being written.

The Asset Helper

This documentation is still being written.

Chapter 39
The YAML Component

The YAML Component loads and dumps YAML files.

What is it?

The Symfony2 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 Symfony2 YAML Component implements the YAML 1.2 version of the specification.

Learn more about the Yaml component in the *The YAML Format* article.

Installation

You can install the component in many different ways:

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

1. `http://yaml.org/`
2. `https://github.com/symfony/Yaml`
3. `https://packagist.org/packages/symfony/yaml`

Why?

Fast

One of the goal of Symfony YAML is to find the right balance between speed and features. It supports just the needed feature to handle configuration 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 Symfony2 YAML Component

The Symfony2 YAML Component is very simple and consists of two main classes: one parses YAML strings (*Parser*[4]), and the other dumps a PHP array to a YAML string (*Dumper*[5]).

On top of these two classes, the *Yaml*[6] class acts as a thin wrapper that simplifies common uses.

Reading YAML Files

The *parse()*[7] method parses a YAML string and converts it to a PHP array:

Listing 39-1
```php
use Symfony\Component\Yaml\Parser;

$yaml = new Parser();

$value = $yaml->parse(file_get_contents('/path/to/file.yml'));
```

4. http://api.symfony.com/2.0/Symfony/Component/Yaml/Parser.html
5. http://api.symfony.com/2.0/Symfony/Component/Yaml/Dumper.html
6. http://api.symfony.com/2.0/Symfony/Component/Yaml/Yaml.html
7. http://api.symfony.com/2.0/Symfony/Component/Yaml/Parser.html#parse()

If an error occurs during parsing, the parser throws a *ParseException*[8] exception indicating the error type and the line in the original YAML string where the error occurred:

Listing 39-2
```php
use Symfony\Component\Yaml\Exception\ParseException;

try {
    $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
} catch (ParseException $e) {
    printf("Unable to parse the YAML string: %s", $e->getMessage());
}
```

As the parser is re-entrant, you can use the same parser object to load different YAML strings.

When loading a YAML file, it is sometimes better to use the *parse()*[9] wrapper method:

Listing 39-3
```php
use Symfony\Component\Yaml\Yaml;

$yaml = Yaml::parse('/path/to/file.yml');
```

The *parse()*[10] static method takes a YAML string or a file containing YAML. Internally, it calls the *parse()*[11] method, but with some added bonuses:

- It executes the YAML file as if it was a PHP file, so that you can embed PHP commands in YAML files;
- When a file cannot be parsed, it automatically adds the file name to the error message, simplifying debugging when your application is loading several YAML files.

Writing YAML Files

The *dump()*[12] method dumps any PHP array to its YAML representation:

Listing 39-4
```php
use Symfony\Component\Yaml\Dumper;

$array = array(
    'foo' => 'bar',
    'bar' => array('foo' => 'bar', 'bar' => 'baz'),
);

$dumper = new Dumper();

$yaml = $dumper->dump($array);

file_put_contents('/path/to/file.yml', $yaml);
```

8. http://api.symfony.com/2.0/Symfony/Component/Yaml/Exception/ParseException.html
9. http://api.symfony.com/2.0/Symfony/Component/Yaml/Yaml.html#parse()
10. http://api.symfony.com/2.0/Symfony/Component/Yaml/Yaml.html#parse()
11. http://api.symfony.com/2.0/Symfony/Component/Yaml/Parser.html#parse()
12. http://api.symfony.com/2.0/Symfony/Component/Yaml/Dumper.html#dump()

Of course, the Symfony2 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*[13] exception.

If you only need to dump one array, you can use the *dump()*[14] static method shortcut:

Listing 39-5
```php
use Symfony\Component\Yaml\Yaml;

$yaml = Yaml::dump($array, $inline);
```

The YAML format supports two kind of representation for arrays, the expanded one, and the inline one. By default, the dumper uses the inline representation:

Listing 39-6
```
{ foo: bar, bar: { foo: bar, bar: baz } }
```

The second argument of the *dump()*[15] method customizes the level at which the output switches from the expanded representation to the inline one:

Listing 39-7
```php
echo $dumper->dump($array, 1);
```

Listing 39-8
```
foo: bar
bar: { foo: bar, bar: baz }
```

Listing 39-9
```php
echo $dumper->dump($array, 2);
```

Listing 39-10
```
foo: bar
bar:
    foo: bar
    bar: baz
```

13. http://api.symfony.com/2.0/Symfony/Component/Yaml/Exception/DumpException.html
14. http://api.symfony.com/2.0/Symfony/Component/Yaml/Yaml.html#dump()
15. http://api.symfony.com/2.0/Symfony/Component/Yaml/Dumper.html#dump()

Chapter 40
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

Listing 40-1
```
A string in YAML
```

Listing 40-2
```
'A singled-quoted string in YAML'
```

In a single quoted string, a single quote `'` must be doubled:

Listing 40-3
```
'A single quote '' in a single-quoted string'
```

Listing 40-4
```
"A double-quoted string in YAML\n"
```

1. http://yaml.org/

Quoted styles are useful when a string starts or ends with one or more relevant spaces.

The double-quoted style provides a way to express arbitrary strings, by using `\` escape sequences. It is very useful when you need to embed a `\n` or a unicode character in a string.

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 40-5
```
|
  \/ /| |\/| |
  / / | |  | |__
```

Alternatively, strings can be written with the folded style, denoted by `>`, where each line break is replaced by a space:

Listing 40-6
```
>
  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 40-7
```
# an integer
12
```

Listing 40-8
```
# an octal
014
```

Listing 40-9
```
# an hexadecimal
0xC
```

Listing 40-10
```
# a float
13.4
```

Listing 40-11
```
# an exponential number
1.2e+34
```

Listing 40-12
```
# 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 40-13
```
2001-12-14t21:59:43.10-05:00
```

Listing 40-14
```
# 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 40-15
```
- PHP
- Perl
- Python
```

The previous YAML file is equivalent to the following PHP code:

Listing 40-16
```
array('PHP', 'Perl', 'Python');
```

Mappings use a colon followed by a space (`: `) to mark each key/value pair:

Listing 40-17
```
PHP: 5.2
MySQL: 5.1
Apache: 2.2.20
```

which is equivalent to this PHP code:

Listing 40-18
```
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 40-19

```
PHP:    5.2
MySQL:  5.1
Apache: 2.2.20
```

YAML uses indentation with one or more spaces to describe nested collections:

Listing 40-20
```yaml
"symfony 1.0":
  PHP:    5.0
  Propel: 1.2
"symfony 1.2":
  PHP:    5.2
  Propel: 1.3
```

The following YAML is equivalent to the following PHP code:

Listing 40-21
```php
array(
  'symfony 1.0' => array(
    'PHP'    => 5.0,
    'Propel' => 1.2,
  ),
  'symfony 1.2' => array(
    'PHP'    => 5.2,
    'Propel' => 1.3,
  ),
);
```

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 40-22
```yaml
'Chapter 1':
  - Introduction
  - Event Types
'Chapter 2':
  - Introduction
  - 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 40-23
```yaml
[PHP, Perl, Python]
```

A mapping can be written as a comma separated list of key/values within curly braces (`{}`):

Listing 40-24
```yaml
{ PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

You can mix and match styles to achieve a better readability:

Listing 40-25
```yaml
'Chapter 1': [Introduction, Event Types]
'Chapter 2': [Introduction, Helpers]
```

Listing 40-26

```
"symfony 1.0": { PHP: 5.0, Propel: 1.2 }
"symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Comments

Comments can be added in YAML by prefixing them with a hash mark (#):

Listing 40-27
```
# Comment on a line
"symfony 1.0": { PHP: 5.0, Propel: 1.2 } # Comment at the end of a line
"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.