Symfony

The Components Book

for Symfony 2.1

generated on October 9, 2012

The Components Book (2.1)

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

Contents at a Glance

Chapter 1
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 PEAR (*pear.symfony.com/ClassLoader*);
- Install it via Composer (*symfony/class-loader* on Packagist).

Usage

New in version 2.1: The `useIncludePath` method was added in Symfony 2.1.

1. `http://symfony.com/PSR0`
2. `http://pear.php.net/manual/en/standards.php`
3. `https://github.com/symfony/ClassLoader`

Registering the *UniversalClassLoader*[4] autoloader is straightforward:

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

use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();

// You can search the include_path as a last resort.
$loader->useIncludePath(true);

// ... 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*[5]:

Listing 1-2
```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()*[6] or *registerNamespaces()*[7] methods:

Listing 1-3
```php
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/symfony/src');

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

$loader->register();
```

For classes that follow the PEAR naming convention, use the *registerPrefix()*[8] or *registerPrefixes()*[9] methods:

Listing 1-4
```php
$loader->registerPrefix('Twig_', __DIR__.'/vendor/twig/twig/lib');

```

4. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/UniversalClassLoader.html
5. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/ApcUniversalClassLoader.html
6. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerNamespace()
7. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerNamespaces()
8. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerPrefix()
9. http://api.symfony.com/2.1/Symfony/Component/ClassLoader/UniversalClassLoader.html#registerPrefixes()

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

$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 1-5
```
$loader->registerNamespaces(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine/common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine/migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine/dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/orm/lib',
));

$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 2
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 PEAR (*pear.symfony.com/Config*);
- Install it via Composer (*symfony/config* on Packagist).

Sections

- *Loading resources*
- *Caching based on resources*
- *Define and process configuration values*

1. `https://github.com/symfony/Config`

Chapter 3
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 3-1
```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 3-2
```php
use Symfony\Component\Config\Loader\FileLoader;
use Symfony\Component\Yaml\Yaml;

```

1. http://api.symfony.com/2.1/Symfony/Component/Config/FileLocator.html
2. http://api.symfony.com/2.1/Symfony/Component/Config/Loader/LoaderInterface.html
3. http://api.symfony.com/2.1/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 3-3

```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/2.1/Symfony/Component/Config/Loader/LoaderResolver.html
5. http://api.symfony.com/2.1/Symfony/Component/Config/Loader/DelegatingLoader.html
6. http://api.symfony.com/2.1/Symfony/Component/Config/Loader/LoaderResolver.html
7. http://api.symfony.com/2.1/Symfony/Component/Config/Loader/LoaderResolver.html

Chapter 4
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 4-1

```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/2.1/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 5
Define and process configuration values

Validate 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 5-1

```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`).

Define 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 5-2
```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;
    }
}
```

Add 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 5-3
```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/2.1/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
2. http://api.symfony.com/2.1/Symfony/Component/Config/Definition/Builder/TreeBuilder.html
3. http://api.symfony.com/2.1/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 5-4
```
$rootNode
    ->arrayNode('connection')
        ->scalarNode('driver')->end()
        ->scalarNode('host')->end()
        ->scalarNode('username')->end()
        ->scalarNode('password')->end()
    ->end()
;
```

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

Listing 5-5
```
$rootNode
    ->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).

An example of this:

Listing 5-6
```
$rootNode
    ->arrayNode('parameters')
        ->isRequired()
        ->requiresAtLeastOneElement()
        ->useAttributeAsKey('name')
```

```
        ->prototype('array')
            ->children()
                ->scalarNode('name')->isRequired()->end()
                ->scalarNode('value')->isRequired()->end()
            ->end()
        ->end()
    ->end()
;
```

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 5-7

```
$rootNode
    ->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()
;
```

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 5-8

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

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

    return $treeBuilder;
}

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

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

```
44      ;
45
46      return $node;
47  }
```

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`.

Another difference between Yaml and XML is in the way arrays of values may be represented. In Yaml you may have:

Listing 5-9
```yaml
twig:
    extensions: ['twig.extension.foo', 'twig.extension.bar']
```

and in XML:

Listing 5-10
```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 5-11
```
$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 5-12
```
$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 5-13

```xml
<connection>default</connection>
<connection>extra</connection>
```

and sometimes only:

Listing 5-14
```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 id is optional in this config:

Listing 5-15
```yaml
connection:
    name: my_mysql_connection
    host: localhost
    driver: mysql
    username: user
    password: pass
```

you can allow the following as well:

Listing 5-16
```yaml
connection: my_mysql_connection
```

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

Listing 5-17
```php
$rootNode
    ->arrayNode('connection')
       ->beforeNormalization()
           ->ifString()
           ->then(function($v) { return array('name'=> $v); })
       ->end()
       ->scalarValue('name')->isRequired()
       // ...
    ->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 5-18
```php
$rootNode
    ->arrayNode('connection')
        ->children()
            ->scalarNode('driver')
                ->isRequired()
                ->validate()
                    ->ifNotInArray(array('mysql', 'sqlite', 'mssql'))
```

4. http://api.symfony.com/2.1/Symfony/Component/Config/Definition/Builder/ExprBuilder.html

```
                    ->thenInvalid('Invalid database driver "%s"')
                ->end()
            ->end()
        ->end()
    ->end()
;
```

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

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

A validation rule also requires a "then" part:

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

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

Processing configuration values

The *`Processor`*[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 5-19

```
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/2.1/Symfony/Component/Config/Definition/Processor.html
6. http://api.symfony.com/2.1/Symfony/Component/Config/Definition/Builder/TreeBuilder.html

Chapter 6
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 PEAR (*pear.symfony.com/Console*);
- Install it via Composer (*symfony/console* on Packagist).

Creating a basic Command

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

Listing 6-1

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

1. `https://github.com/symfony/Console`

```
12      {
13          $this
14              ->setName('demo:greet')
15              ->setDescription('Greet someone')
16              ->addArgument(
17                  'name',
18                  InputArgument::OPTIONAL,
19                  'Who do you want to greet?'
20              )
21              ->addOption(
22                 'yell',
23                 null,
24                 InputOption::VALUE_NONE,
25                 'If set, the task will yell in uppercase letters'
26              )
27          ;
28      }
29
30      protected function execute(InputInterface $input, OutputInterface $output)
31      {
32          $name = $input->getArgument('name');
33          if ($name) {
34              $text = 'Hello '.$name;
35          } else {
36              $text = 'Hello';
37          }
38
39          if ($input->getOption('yell')) {
40              $text = strtoupper($text);
41          }
42
43          $output->writeln($text);
44      }
45  }
```

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

Listing 6-2

```
1  #!/usr/bin/env php
2  # app/console
3  <?php
4
5  use Acme\DemoBundle\Command\GreetCommand;
6  use Symfony\Component\Console\Application;
7
8  $application = new Application();
9  $application->add(new GreetCommand);
10 $application->run();
```

Test the new console command by running the following

Listing 6-3

```
1  $ app/console demo:greet Fabien
```

This will print the following to the command line:

Listing 6-4

```
1  Hello Fabien
```

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

Listing 6-5
```
$ app/console demo:greet Fabien --yell
```

This prints:

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

Coloring the Output

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

Listing 6-7
```
// 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*[2]:

Listing 6-8
```
$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`.

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 6-9
```
$this
    // ...
    ->addArgument(
        'name',
        InputArgument::REQUIRED,
        'Who do you want to greet?'
    )
    ->addArgument(
        'last_name',
        InputArgument::OPTIONAL,
```

2. http://api.symfony.com/2.1/Symfony/Component/Console/Formatter/OutputFormatterStyle.html

```
11          'Your last name?'
12      );
```

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

Listing 6-10
```
1  if ($lastName = $input->getArgument('last_name')) {
2      $text .= ' '.$lastName;
3  }
```

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

Listing 6-11
```
1  $ app/console demo:greet Fabien
2  $ 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 6-12
```
1  $this
2      // ...
3      ->addOption(
4          'iterations',
5          null,
6          InputOption::VALUE_REQUIRED,
7          'How many times should the message be printed?',
8          1
9      );
```

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

Listing 6-13
```
1  for ($i = 0; $i < $input->getOption('iterations'); $i++) {
2      $output->writeln($text);
3  }
```

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

Listing 6-14
```
1  $ app/console demo:greet Fabien
2  $ 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 6-15
```
$ 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 6-16
```
$this
    // ...
    ->addOption(
        'iterations',
        null,
        InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY,
        'How many times should the message be printed?',
        1
    );
```

Asking the User for Information

When creating commands, you have the ability to collect more information from the user by asking him/her questions. For example, suppose you want to confirm an action before actually executing it. Add the following to your command:

Listing 6-17
```
$dialog = $this->getHelperSet()->get('dialog');
if (!$dialog->askConfirmation(
        $output,
        '<question>Continue with this action?</question>',
        false
    )) {
    return;
}
```

In this case, the user will be asked "Continue with this action", and unless they answer with `y`, the task will stop running. The third argument to `askConfirmation` is the default value to return if the user doesn't enter any input.

You can also ask questions with more than a simple yes/no answer. For example, if you needed to know the name of something, you might do the following:

Listing 6-18

```php
$dialog = $this->getHelperSet()->get('dialog');
$name = $dialog->ask(
    $output,
    'Please enter the name of the widget',
    'foo'
);
```

Testing Commands

Symfony2 provides several tools to help you test your commands. The most useful one is the *CommandTester*[3] class. It uses special input and output classes to ease testing without a real console:

Listing 6-19

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

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

        // ...
    }
}
```

The *getDisplay()*[4] 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()*[5] method:

Listing 6-20

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

3. http://api.symfony.com/2.1/Symfony/Component/Console/Tester/CommandTester.html
4. http://api.symfony.com/2.1/Symfony/Component/Console/Tester/CommandTester.html#getDisplay()
5. http://api.symfony.com/2.1/Symfony/Component/Console/Tester/CommandTester.html#execute()

```
13
14          $command = $application->find('demo:greet');
15          $commandTester = new CommandTester($command);
16          $commandTester->execute(
17              array('command' => $command->getName(), 'name' => 'Fabien')
18          );
19
20          $this->assertRegExp('/Fabien/', $commandTester->getDisplay());
21      }
22  }
```

You can also test a whole console application by using *ApplicationTester*[6].

Calling an existing Command

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

Calling a command from another one is straightforward:

Listing 6-21

```
1  protected function execute(InputInterface $input, OutputInterface $output)
2  {
3      $command = $this->getApplication()->find('demo:greet');
4
5      $arguments = array(
6          'command' => 'demo:greet',
7          'name'    => 'Fabien',
8          '--yell'  => true,
9      );
10
11     $input = new ArrayInput($arguments);
12     $returnCode = $command->run($input, $output);
13
14     // ...
15 }
```

First, you *find()*[7] the command you want to execute by passing the command name.

Then, you need to create a new *ArrayInput*[8] 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).

6. http://api.symfony.com/2.1/Symfony/Component/Console/Tester/ApplicationTester.html
7. http://api.symfony.com/2.1/Symfony/Component/Console/Application.html#find()
8. http://api.symfony.com/2.1/Symfony/Component/Console/Input/ArrayInput.html

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.

Chapter 7
Console Usage

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 7-1
```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 7-2
```
$ php app/console list
```

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

Listing 7-3
```
$ 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 7-4
```
$ php app/console help list
```

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

Listing 7-5
```
$ 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 7-6
```
$ php app/console list --help
$ php app/console list -h
```

You can suppress output with:

Listing 7-7
```
$ 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 7-8
```
$ 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 7-9
```
$application = new Application('Acme Console Application', '1.2');
```

then you can use:

Listing 7-10
```
$ php app/console list --version
$ php app/console list -V
```

to get this information output:

Listing 7-11
```
Acme Console Application version 1.2
```

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

Listing 7-12
```
console tool
```

You can force turning on ANSI output coloring with:

Listing 7-13
```
$ php app/console list --ansi
```

or turn it off with:

Listing 7-14
```
$ php app/console list --no-ansi
```

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

Listing 7-15

```
$ 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 7-16
```
$ 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 7-17
```
$ 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 8
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 PEAR (*pear.symfony.com/CssSelector*);
- Install it via Composer (*symfony/css-selector* on Packagist).

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`

The `CssSelector` component

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

Listing 8-1
```php
use Symfony\Component\CssSelector\CssSelector;

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

This gives the following output:

Listing 8-2
```
descendant-or-self::div[contains(concat(' ',normalize-space(@class), ' '), ' item ')]/h4/a
```

You can use this expression with, for instance, *`DOMXPath`*[2] or *`SimpleXMLElement`*[3] to find elements in a document.

The *`Crawler::filter()`*[4] 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 `*`.

2. http://php.net/manual/en/class.domxpath.php
3. http://php.net/manual/en/class.simplexmlelement.php
4. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Crawler.html#filter()

Chapter 9
The DomCrawler Component

The DomCrawler Component eases DOM navigation for HTML and XML documents.

Installation

You can install the component in many different ways:

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

Usage

The *`Crawler`*[2] class provides methods to query and manipulate HTML and XML documents.

An instance of the Crawler represents a set (*`SplObjectStorage`*[3]) of *`DOMElement`*[4] objects, which are basically nodes that you can traverse easily:

Listing 9-1

```
use Symfony\Component\DomCrawler\Crawler;

$html = <<<'HTML'
<!DOCTYPE html>
<html>
    <body>
        <p class="message">Hello World!</p>
        <p>Hello Crawler!</p>
    </body>
</html>
```

1. https://github.com/symfony/DomCrawler
2. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Crawler.html
3. http://php.net/manual/en/class.splobjectstorage.php
4. http://php.net/manual/en/class.domelement.php

```
HTML;

$crawler = new Crawler($html);

foreach ($crawler as $domElement) {
    print $domElement->nodeName;
}
```

Specialized *Link*[5] and *Form*[6] 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 9-2
```
$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 9-3
```
$crawler = $crawler->filter('body > p');
```

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

Listing 9-4
```
$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*[7] instance with filtered content.

Node Traversing

Access node by its position on the list:

Listing 9-5
```
$crawler->filter('body > p')->eq(0);
```

Get the first or last node of the current selection:

Listing 9-6

5. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Link.html
6. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Form.html
7. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Crawler.html

```php
$crawler->filter('body > p')->first();
$crawler->filter('body > p')->last();
```

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

Listing 9-7
```php
$crawler->filter('body > p')->siblings();
```

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

Listing 9-8
```php
$crawler->filter('body > p')->nextAll();
$crawler->filter('body > p')->previousAll();
```

Get all the child or parent nodes:

Listing 9-9
```php
$crawler->filter('body')->children();
$crawler->filter('body > p')->parents();
```

All the traversal methods return a new *Crawler*[8] instance.

Accessing Node Values

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

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

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

Listing 9-11
```php
$class = $crawler->filterXPath('//body/p')->attr('class');
```

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

Listing 9-12
```php
$attributes = $crawler->filterXpath('//body/p')->extract(array('_text', 'class'));
```

Special attribute `_text` represents a node value.

Call an anonymous function on each node of the list:

Listing 9-13
```php
$nodeValues = $crawler->filter('p')->each(function ($node, $i) {
    return $node->nodeValue;
});
```

The anonymous function receives the position and the node as arguments. The result is an array of values returned by the anonymous function calls.

8. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Crawler.html

Adding the Content

The crawler supports multiple ways of adding the content:

Listing 9-14
```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>');
```

As the Crawler's implementation is based on the DOM extension, it is also able to interact with native *DOMDocument*[9], *DOMNodeList*[10] and *DOMNode*[11] objects:

Listing 9-15
```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);
```

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 us a special *Link*[12] object:

Listing 9-16
```php
$linksCrawler = $crawler->selectLink('Go elsewhere...');
$link = $linksCrawler->link();

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

The *Link*[13] object has several useful methods to get more information about the selected link itself:

Listing 9-17
```php
// return the raw href value
$href = $link->getRawUri();

```

9. http://php.net/manual/en/class.domdocument.php
10. http://php.net/manual/en/class.domnodelist.php
11. http://php.net/manual/en/class.domnode.php
12. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Link.html
13. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Link.html

```
// return the proper URI that can be used to make another request
$uri = $link->getUri();
```

The `getUri()` is especially useful as it cleans the `href` value and transforms it into how it should really be processed. For example, for a link with `href="#foo"`, this would return the full URI of the current page suffixed with `#foo`. The return from `getUri()` is always a full URI that you can act on.

Forms

Special treatment is also given to forms. A `selectButton()` method is available on the Crawler which returns another Crawler that matches a button (`input[type=submit]`, `input[type=image]`, or a `button`) with the given text. This method is especially useful because you can use it to return a *Form*[14] object that represents the form that the button lives in:

Listing 9-18
```
$form = $crawler->selectButton('validate')->form();

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

The *Form*[15] object has lots of very useful methods for working with forms:

Listing 9-19
```
$uri = $form->getUri();

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

The *getUri()*[16] 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 9-20
```
// 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 9-21
```
<form>
    <input name="multi[]" />
    <input name="multi[]" />
    <input name="multi[dimensional]" />
</form>
```

14. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Form.html
15. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Form.html
16. http://api.symfony.com/2.1/Symfony/Component/DomCrawler/Form.html#getUri()

You must specify the fully qualified name of the field:

Listing 9-22
```
// Set a single field
$form->setValue('multi[0]', 'value');

// Set multiple fields at once
$form->setValue('multi', array(
    1             => 'value',
    'dimensional' => 'an other value'
));
```

This is great, but it gets better! The `Form` object allows you to interact with your form like a browser, selecting radio values, ticking checkboxes, and uploading files:

Listing 9-23
```
$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 9-24
```
$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 9-25
```
$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*[17]. Goutte understands the Symfony Crawler object and can use it to submit forms directly:

Listing 9-26
```
use Goutte\Client;

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

```

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

```php
// select the form and fill in some values
$form = $crawler->selectButton('Log in')->form();
$form['login'] = 'symfonyfan';
$form['password'] = 'anypass';

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

Chapter 10
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 PEAR (*pear.symfony.com/DependencyInjection*);
- Install it via Composer (*symfony/dependency-injection* on Packagist).

Basic Usage

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

Listing 10-1

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

You can register this in the container as a service:

1. https://github.com/symfony/DependencyInjection

Listing 10-2
```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 10-3
```php
class Mailer
{
    private $transport;

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

    // ...
}
```

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

Listing 10-4
```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 10-5
```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 10-6
```php
use Mailer;

class NewsletterManager
{
    private $mailer;

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

```

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

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

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

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

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

Listing 10-10
```
1  use Symfony\Component\DependencyInjection\ContainerBuilder;
2  use Symfony\Component\DependencyInjection\Reference;
3
4  $container = new ContainerBuilder();
5
```

```php
// ...

$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. To do this you also need to install *the Config Component*.

Loading an XML config file:

Listing 10-11
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;

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

Loading a YAML config file:

Listing 10-12
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

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

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

The `newsletter_manager` and `mailer` services can be set up using config files:

Listing 10-13
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    mailer.transport: sendmail

services:
    mailer:
```

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

Chapter 11
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 11-1
```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 11-2
```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 11-3
```php
class NewsletterManager
{
    protected $mailer;

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

    // ...
}
```

Listing 11-4
```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 11-5
```php
class NewsletterManager
{
    public $mailer;

    // ...
}
```

Listing 11-6
```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 12

Working with Container Parameters and Definitions

Getting and Setting Container Parameters

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

Listing 12-1
```
1 $container->hasParameter($name);
```

You can retrieve parameters set in the container with:

Listing 12-2
```
1 $container->getParameter($name);
```

and set a parameter in the container with:

Listing 12-3
```
1 $container->setParameter($name, $value);
```

Getting and Setting Service Definitions

There are also some helpful methods for working with the service definitions.
To find out if there is a definition for a service id:

Listing 12-4
```
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 12-5
```php
$container->getDefinition($serviceId);
```

or:

Listing 12-6
```php
$container->findDefinition($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 12-7
```php
$container->setDefinition($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 12-8
```php
$definition->getClass();
```

and to set a different class:

Listing 12-9
```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 12-10
```php
$definition->getArguments();
```

or to get a single argument by its position:

Listing 12-11
```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 12-12

1. http://api.symfony.com/2.1/Symfony/Component/DependencyInjection/Definition.html

```
1 $definition->addArgument($argument);
```

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

Listing 12-13
```
1 use Symfony\Component\DependencyInjection\Reference;
2
3 // ...
4
5 $definition->addArgument(new Reference('service_id'));
```

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

Listing 12-14
```
1 $definition->replaceArgument($index, $argument);
```

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

Listing 12-15
```
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 12-16
```
1 $definition->getMethodCalls();
```

Add a method call with:

Listing 12-17
```
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 12-18
```
1 $definition->setMethodCalls($methodCalls);
```

Chapter 13
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 13-1
```
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 13-2
```
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 Parameters and Definitions*.

1. http://api.symfony.com/2.1/Symfony/Component/DependencyInjection/Compiler/CheckDefinitionValidityPass.html
2. http://api.symfony.com/2.1/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 13-3
```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 13-4
```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 13-5
```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 13-6

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

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

$container->registerExtension(new AcmeDemoExtension);
// ...
$container->compile();
```

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

Listing 13-7
```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 13-8
```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 13-9
```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

    // ...
}
```

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 13-10

```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 13-11

```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 13-12

```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'])

    // ...
}
```

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 13-13

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

    $loader = new XmlFileLoader(
        $container,
        new FileLocator(__DIR__.'/../Resources/config')
    );
    $loader->load('services.xml');

    if ($config['advanced']) {
        $loader->load('advanced.xml');
    }
}
```

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 implements the *`CompilerPassInterface`*[3] interface. The compiler pass gives you an opportunity to manipulate the service definitions that have been compiled. This can be very powerful, but is not something needed in everyday use.

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

Listing 13-14

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

The container's parameters and definitions can be manipulated using the methods described in the *Working with Container Parameters and 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 13-15

3. http://api.symfony.com/2.1/Symfony/Component/DependencyInjection/Compiler/CompilerPassInterface.html

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

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

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

Controlling the Pass Ordering

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

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

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

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

Listing 13-16
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\PassConfig;

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

Dumping the Configuration for Performance

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

Listing 13-17
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Dumper\PhpDumper

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

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

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

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

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

Listing 13-18

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

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

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

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

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

Listing 13-19

```
// ...

// 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 13-20

```
// ...

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

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

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

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

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

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

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

Chapter 14
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 14-1
```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 14-2
```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 14-3
```
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 14-4
```
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 14-5
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

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

Compiler passes are registered differently is 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 14-6
```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 14-7
```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 14-8

```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 15
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 15-1
```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 15-2
```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 15-3
```yaml
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 15-4
```yaml
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 16

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 16-1

```
1  class NewsletterManager
2  {
3      protected $mailer;
4      protected $emailFormatter;
5
6      public function setMailer(Mailer $mailer)
7      {
8          $this->mailer = $mailer;
9      }
10
11     public function setEmailFormatter(EmailFormatter $emailFormatter)
12     {
13         $this->emailFormatter = $emailFormatter;
14     }
15
16     // ...
17 }
```

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

Listing 16-2

```
1  class GreetingCardManager
2  {
3      protected $mailer;
4      protected $emailFormatter;
5
6      public function setMailer(Mailer $mailer)
7      {
8          $this->mailer = $mailer;
```

```
    }

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

    // ...
}
```

The service config for these classes would look something like this:

Listing 16-3
```
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 16-4
```
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 16-5
```php
class NewsletterManager extends MailManager
{
    // ...
}
```

and:

Listing 16-6
```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 16-7
```yaml
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    greeting_card_manager.class: GreetingCardManager
    mail_manager.class: MailManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     "%mail_manager.class%"
        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 parent class is abstract as it should not be directly instantiated. Setting it to abstract in the config file as has been done above will mean that it can only be used as a parent service and cannot be used directly as a service to inject and will be removed at compile time. In other words, it exists merely as a "template" that other services can use.

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 16-8

```
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    greeting_card_manager.class: GreetingCardManager
    mail_manager.class: MailManager
services:
    my_mailer:
        # ...
    my_alternative_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     "%mail_manager.class%"
        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 16-9

```php
abstract class MailManager
{
    protected $filters;

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

    // ...
}
```

If you had the following config:

Listing 16-10
```yaml
parameters:
    # ...
    newsletter_manager.class: NewsletterManager
    mail_manager.class: MailManager
services:
    my_filter:
        # ...
    another_filter:
        # ...
    mail_manager:
        class:     "%mail_manager.class%"
        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`.

Chapter 17
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 17-1
```
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 only one other service, this will result in an inlined instantiation (e.g. `new PrivateFooBar()`) inside this other service, making it publicly unavailable at runtime.

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

Here is an example:

Listing 17-2
```
1 services:
2    foo:
3      class: Example\Foo
4      public: false
```

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

Listing 17-3
```
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 17-4
```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 17-5
```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 17-6
```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 18
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. 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 19
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 *Symfony2 HttpKernel component*[2]. 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.

1. `http://en.wikipedia.org/wiki/Observer_pattern`
2. `https://github.com/symfony/HttpKernel`

Installation

You can install the component in many different ways:

- Use the official Git repository (*https://github.com/symfony/EventDispatcher*[3]);
- Install it via PEAR (*pear.symfony.com/EventDispatcher*);
- Install it via Composer (*symfony/event-dispatcher* on Packagist).

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 19-1

3. https://github.com/symfony/EventDispatcher
4. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/2.1/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html

```php
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
```

Connecting Listeners

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

Listing 19-2
```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 19-3
```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 19-4
```php
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

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

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

In many cases, a special `Event` subclass that's specific to the given event is passed to the listener. This gives the listener access to special information about the event. Check the documentation or implementation of each event to determine the exact `Symfony\Component\EventDispatcher\Event` instance that's being passed. For example, the `kernel.event` event passes an instance of `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

Listing 19-5
```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 19-6
```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 19-7
```php
namespace Acme\StoreBundle\Event;

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 19-8
```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 19-9
```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
}
```

8. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventDispatcher.html#dispatch()

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 19-10
```php
namespace Acme\StoreBundle\Event;

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

class StoreSubscriber implements EventSubscriberInterface
{
    static public function getSubscribedEvents()
    {
        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 19-11
```php
use Acme\StoreBundle\Event\StoreSubscriber;

```

9. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventSubscriberInterface.html
10. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventDispatcher.html#addSubscriber()

```
3 $subscriber = new StoreSubscriber();
4 $dispatcher->addSubscriber($subscriber);
```

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

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:

Listing 19-12
```
1 use Acme\StoreBundle\Event\FilterOrderEvent;
2
3 public function onStoreOrder(FilterOrderEvent $event)
4 {
5     // ...
6
7     $event->stopPropagation();
8 }
```

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

It is possible to detect if an event was stopped by using the *`isPropagationStopped()`*[12] method which returns a boolean value:

Listing 19-13
```
1 $dispatcher->dispatch('foo.event', $event);
2 if ($event->isPropagationStopped()) {
3     // ...
4 }
```

EventDispatcher aware Events and Listeners

New in version 2.1: The `Event` object contains a reference to the invoking dispatcher since Symfony 2.1

The `EventDispatcher` always injects a reference to itself in the passed event object. This means that all listeners have direct access to the `EventDispatcher` object that notified the listener via the passed `Event` object's *`getDispatcher()`*[13] method.

This can lead to some advanced applications of the `EventDispatcher` including letting listeners dispatch other events, event chaining or even lazy loading of more listeners into the dispatcher object. Examples follow:

Lazy loading listeners:

11. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html#stopPropagation()
12. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html#isPropagationStopped()
13. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html#getDispatcher()

Listing 19-14
```php
use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Event\StoreSubscriber;

class Foo
{
    private $started = false;

    public function myLazyListener(Event $event)
    {
        if (false === $this->started) {
            $subscriber = new StoreSubscriber();
            $event->getDispatcher()->addSubscriber($subscriber);
        }

        $this->started = true;

        // ... more code
    }
}
```

Dispatching another event from within a listener:

Listing 19-15
```php
use Symfony\Component\EventDispatcher\Event;

class Foo
{
    public function myFooListener(Event $event)
    {
        $event->getDispatcher()->dispatch('log', $event);

        // ... more code
    }
}
```

While this above is sufficient for most uses, if your application uses multiple `EventDispatcher` instances, you might need to specifically inject a known instance of the `EventDispatcher` into your listeners. This could be done using constructor or setter injection as follows:

Constructor injection:

Listing 19-16
```php
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class Foo
{
    protected $dispatcher = null;

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

Or setter injection:

Listing 19-17
```php
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class Foo
```

```
4  {
5      protected $dispatcher = null;
6
7      public function setEventDispatcher(EventDispatcherInterface $dispatcher)
8      {
9          $this->dispatcher = $dispatcher;
10     }
11 }
```

Choosing between the two is really a matter of taste. Many tend to prefer the constructor injection as the objects are fully initialized at construction time. But when you have a long list of dependencies, using setter injection can be the way to go, especially for optional dependencies.

Dispatcher Shortcuts

New in version 2.1: `EventDispatcher::dispatch()` method returns the event since Symfony 2.1.

The *`EventDispatcher::dispatch`*[14] method always returns an *`Event`*[15] object. This allows for various shortcuts. For example if one does not need a custom event object, one can simply rely on a plain *`Event`*[16] object. You do not even need to pass this to the dispatcher as it will create one by default unless you specifically pass one:

Listing 19-18
```
1 $dispatcher->dispatch('foo.event');
```

Moreover, the EventDispatcher always returns whichever event object that was dispatched, i.e. either the event that was passed or the event that was created internally by the dispatcher. This allows for nice shortcuts:

Listing 19-19
```
1 if (!$dispatcher->dispatch('foo.event')->isPropagationStopped()) {
2     // ...
3 }
```

Or:

Listing 19-20
```
1 $barEvent = new BarEvent();
2 $bar = $dispatcher->dispatch('bar.event', $barEvent)->getBar();
```

Or:

Listing 19-21
```
1 $response = $dispatcher->dispatch('bar.event', new BarEvent())->getBar();
```

and so on...

14. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventDispatcher.html#dispatch()
15. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html
16. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html

Event Name Introspection

New in version 2.1: Added event name to the `Event` object since Symfony 2.1

Since the `EventDispatcher` already knows the name of the event when dispatching it, the event name is also injected into the *`Event`*[17] objects, making it available to event listeners via the *`getName()`*[18] method.

The event name, (as with any other data in a custom event object) can be used as part of the listener's processing logic:

Listing 19-22

```php
use Symfony\Component\EventDispatcher\Event;

class Foo
{
    public function myEventListener(Event $event)
    {
        echo $event->getName();
    }
}
```

17. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html
18. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html#getName()

Chapter 20
The Generic Event Object

New in version 2.1: The `GenericEvent` event class was added in Symfony 2.1

The base *`Event`*[1] class provided by the `Event Dispatcher` component is deliberately sparse to allow the creation of API specific event objects by inheritance using OOP. This allow for elegant and readable code in complex applications.

The *`GenericEvent`*[2] is available for convenience for those who wish to use just one event object throughout their application. It is suitable for most purposes straight out of the box, because it follows the standard observer pattern where the event object encapsulates an event 'subject', but has the addition of optional extra arguments.

`GenericEvent`[3] has a simple API in addition to the base class *`Event`*[4]

- *`__construct()`*[5]: Constructor takes the event subject and any arguments;
- *`getSubject()`*[6]: Get the subject;
- *`setArgument()`*[7]: Sets an argument by key;
- *`setArguments()`*[8]: Sets arguments array;
- *`getArgument()`*[9]: Gets an argument by key;
- *`getArguments()`*[10]: Getter for all arguments;
- *`hasArgument()`*[11]: Returns true if the argument key exists;

1. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html
2. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html
3. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html
4. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/Event.html
5. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#__construct()
6. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#getSubject()
7. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#setArgument()
8. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#setArguments()
9. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#getArgument()
10. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#getArguments()
11. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/GenericEvent.html#hasArgument()

The `GenericEvent` also implements *`ArrayAccess`*[12] on the event arguments which makes it very convenient to pass extra arguments regarding the event subject.

The following examples show use-cases to give a general idea of the flexibility. The examples assume event listeners have been added to the dispatcher.

Simply passing a subject:

Listing 20-1
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = GenericEvent($subject);
$dispatcher->dispatch('foo', $event);

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if ($event->getSubject() instanceof Foo) {
            // ...
        }
    }
}
```

Passing and processing arguments using the *`ArrayAccess`*[13] API to access the event arguments:

Listing 20-2
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject, array('type' => 'foo', 'counter' => 0));
$dispatcher->dispatch('foo', $event);

echo $event['counter'];

class FooListener
{
    public function handler(GenericEvent $event)
    {
        if (isset($event['type']) && $event['type'] === 'foo') {
            // ... do something
        }

        $event['counter']++;
    }
}
```

Filtering data:

Listing 20-3
```php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent($subject, array('data' => 'foo'));
$dispatcher->dispatch('foo', $event);

echo $event['data'];

class FooListener
{
    public function filter(GenericEvent $event)
```

12. http://php.net/manual/en/class.arrayaccess.php
13. http://php.net/manual/en/class.arrayaccess.php

```
11      {
12          strtolower($event['data']);
13      }
14  }
```

Chapter 21
The Container Aware Event Dispatcher

New in version 2.1: This feature was moved into the EventDispatcher component in Symfony 2.1.

Introduction

The *`ContainerAwareEventDispatcher`*[1] is a special event dispatcher implementation which is coupled to the service container that is part of *the Dependency Injection component*. It allows services to be specified as event listeners making the event dispatcher extremely powerful.

Services are lazy loaded meaning the services attached as listeners will only be created if an event is dispatched that requires those listeners.

Setup

Setup is straightforward by injecting a *`ContainerInterface`*[2] into the *`ContainerAwareEventDispatcher`*[3]:

Listing 21-1
```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher;

$container = new ContainerBuilder();
$dispatcher = new ContainerAwareEventDispatcher($container);
```

1. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html
2. http://api.symfony.com/2.1/Symfony/Component/DependencyInjection/ContainerInterface.html
3. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html

Adding Listeners

The *Container Aware Event Dispatcher* can either load specified services directly, or services that implement *`EventSubscriberInterface`*[4].

The following examples assume the service container has been loaded with any services that are mentioned.

Services must be marked as public in the container.

Adding Services

To connect existing service definitions, use the *`addListenerService()`*[5] method where the `$callback` is an array of `array($serviceId, $methodName)`:

Listing 21-2
```
1 $dispatcher->addListenerService($eventName, array('foo', 'logListener'));
```

Adding Subscriber Services

`EventSubscribers` can be added using the *`addSubscriberService()`*[6] method where the first argument is the service ID of the subscriber service, and the second argument is the the service's class name (which must implement *`EventSubscriberInterface`*[7]) as follows:

Listing 21-3
```
1 $dispatcher->addSubscriberService('kernel.store_subscriber', 'StoreSubscriber');
```

The `EventSubscriberInterface` will be exactly as you would expect:

Listing 21-4
```
1  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
2  // ...
3
4  class StoreSubscriber implements EventSubscriberInterface
5  {
6      static public function getSubscribedEvents()
7      {
8          return array(
9              'kernel.response' => array(
10                 array('onKernelResponsePre', 10),
11                 array('onKernelResponsePost', 0),
12             ),
13             'store.order'     => array('onStoreOrder', 0),
14         );
15     }
16
17     public function onKernelResponsePre(FilterResponseEvent $event)
18     {
19         // ...
20     }
```

4. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventSubscriberInterface.html
5. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#addListenerService()
6. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/ContainerAwareEventDispatcher.html#addSubscriberService()
7. http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventSubscriberInterface.html

```
21
22      public function onKernelResponsePost(FilterResponseEvent $event)
23      {
24          // ...
25      }
26
27      public function onStoreOrder(FilterOrderEvent $event)
28      {
29          // ...
30      }
31  }
```

Chapter 22
The Filesystem Component

The Filesystem components provides basic utilities for the filesystem.

New in version 2.1: The Filesystem Component is new to Symfony 2.1. Previously, the `Filesystem` class was located in the `HttpKernel` component.

Installation

You can install the component in many different ways:

- Use the official Git repository (*https://github.com/symfony/Filesystem*[1]);
- Install it via PEAR (*pear.symfony.com/Filesystem*);
- Install it via Composer (*symfony/filesystem* on Packagist).

Usage

The *`Filesystem`*[2] class is the unique endpoint for filesystem operations:

Listing 22-1
```php
use Symfony\Component\Filesystem\Filesystem;
use Symfony\Component\Filesystem\Exception\IOException;

$fs = new Filesystem();

try {
    $fs->mkdir('/tmp/random/dir/' . mt_rand());
} catch (IOException $e) {
```

1. https://github.com/symfony/Filesystem
2. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html

```
 9      echo "An error occured while creating your directory";
10  }
```

Methods *mkdir()*[3], *chown()*[4], *chgrp()*[5], *chown()*[6], *remove()*[7] and *touch()*[8] can receive a string, an array or any object implementing *Traversable*[9] as the target argument.

Mkdir

Mkdir creates directory. On posix filesystems, directories are created with a default mode value *0777*. You can use the second argument to set your own mode:

Listing 22-2
```
1  $fs->mkdir('/tmp/photos', 0700);
```

You can pass an array or any *Traversable*[10] object as the first argument.

Exists

Exists checks for the presence of all files or directories and returns false if a file is missing:

Listing 22-3
```
1  // this directory exists, return true
2  $fs->exists('/tmp/photos');
3
4  // rabbit.jpg exists, bottle.png does not exists, return false
5  $fs->exists(array('rabbit.jpg', 'bottle.png'));
```

You can pass an array or any *Traversable*[11] object as the first argument.

Copy

This method is used to copy files. If the target already exists, the file is copied only if the source modification date is earlier than the target. This behavior can be overridden by the third boolean argument:

Listing 22-4

3. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#mkdir()
4. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#chown()
5. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#chgrp()
6. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#chown()
7. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#remove()
8. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#touch()
9. http://php.net/manual/en/class.traversable.php
10. http://php.net/manual/en/class.traversable.php
11. http://php.net/manual/en/class.traversable.php

```php
// works only if image-ICC has been modified after image.jpg
$fs->copy('image-ICC.jpg', 'image.jpg');

// image.jpg will be overridden
$fs->copy('image-ICC.jpg', 'image.jpg', true);
```

Touch

Touch sets access and modification time for a file. The current time is used by default. You can set your own with the second argument. The third argument is the access time:

Listing 22-5
```php
// set modification time to the current timestamp
$fs->touch('file.txt');
// set modification time 10 seconds in the future
$fs->touch('file.txt', time() + 10);
// set access time 10 seconds in the past
$fs->touch('file.txt', time(), time() - 10);
```

You can pass an array or any *Traversable*[12] object as the first argument.

Chown

Chown is used to change the owner of a file. The third argument is a boolean recursive option:

Listing 22-6
```php
// set the owner of the lolcat video to www-data
$fs->chown('lolcat.mp4', 'www-data');
// change the owner of the video directory recursively
$fs->chown('/video', 'www-data', true);
```

You can pass an array or any *Traversable*[13] object as the first argument.

Chgrp

Chgrp is used to change the group of a file. The third argument is a boolean recursive option:

Listing 22-7
```php
// set the group of the lolcat video to nginx
$fs->chgrp('lolcat.mp4', 'nginx');
// change the group of the video directory recursively
$fs->chgrp('/video', 'nginx', true);
```

12. http://php.net/manual/en/class.traversable.php
13. http://php.net/manual/en/class.traversable.php

You can pass an array or any *Traversable*[14] object as the first argument.

Chmod

Chmod is used to change the mode of a file. The third argument is a boolean recursive option:

Listing 22-8
```php
// set the mode of the video to 0600
$fs->chmod('video.ogg', 0600);
// change the mod of the src directory recursively
$fs->chmod('src', 0700, true);
```

You can pass an array or any *Traversable*[15] object as the first argument.

Remove

Remove let's you remove files, symlink, directories easily:

Listing 22-9
```php
$fs->remove(array('symlink', '/path/to/directory', 'activity.log'));
```

You can pass an array or any *Traversable*[16] object as the first argument.

Rename

Rename is used to rename files and directories:

Listing 22-10
```php
//rename a file
$fs->rename('/tmp/processed_video.ogg', '/path/to/store/video_647.ogg');
//rename a directory
$fs->rename('/tmp/files', '/path/to/store/files');
```

symlink

Creates a symbolic link from the target to the destination. If the filesystem does not support symbolic links, a third boolean argument is available:

Listing 22-11
```php
// create a symbolic link
$fs->symlink('/path/to/source', '/path/to/destination');
// duplicate the source directory if the filesystem does not support symbolic links
$fs->symlink('/path/to/source', '/path/to/destination', true);
```

14. http://php.net/manual/en/class.traversable.php
15. http://php.net/manual/en/class.traversable.php
16. http://php.net/manual/en/class.traversable.php

makePathRelative

Return the relative path of a directory given another one:

Listing 22-12
```
// returns '../'
$fs->makePathRelative('/var/lib/symfony/src/Symfony/', '/var/lib/symfony/src/Symfony/
Component');
// returns 'videos'
$fs->makePathRelative('/tmp', '/tmp/videos');
```

mirror

Mirrors a directory:

Listing 22-13
```
$fs->mirror('/path/to/source', '/path/to/target');
```

isAbsolutePath

isAbsolutePath returns true if the given path is absolute, false otherwise:

Listing 22-14
```
// return true
$fs->isAbsolutePath('/tmp');
// return true
$fs->isAbsolutePath('c:\\Windows');
// return false
$fs->isAbsolutePath('tmp');
// return false
$fs->isAbsolutePath('../dir');
```

Error Handling

Whenever something wrong happens, an exception implementing *ExceptionInterface*[17] is thrown.

Prior to version 2.1, *mkdir()*[18] returned a boolean and did not throw exceptions. As of 2.1, a *IOException*[19] is thrown if a directory creation fails.

17. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Exception/ExceptionInterface.html
18. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Filesystem.html#mkdir()
19. http://api.symfony.com/2.1/Symfony/Component/Filesystem/Exception/IOException.html

Chapter 23
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 PEAR (*pear.symfony.com/Finder*);
- Install it via Composer (*symfony/finder* on Packagist).

Usage

The *Finder*[2] class finds files and/or directories:

Listing 23-1
```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. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html

The `$file` is an instance of *SplFileInfo*[3] which extends *SplFileInfo*[4] 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*[5]. So, instead of iterating over the Finder with `foreach`, you can also convert it to an array with the *iterator_to_array*[6] method, or get the number of items with *iterator_count*[7].

Criteria

Location

The location is the only mandatory criteria. It tells the finder which directory to use for the search:

Listing 23-2
```php
$finder->in(__DIR__);
```

Search in several locations by chaining calls to *in()*[8]:

Listing 23-3
```php
$finder->files()->in(__DIR__)->in('/elsewhere');
```

Exclude directories from matching with the *exclude()*[9] method:

Listing 23-4
```php
$finder->in(__DIR__)->exclude('ruby');
```

As the Finder uses PHP iterators, you can pass any URL with a supported *protocol*[10]:

Listing 23-5
```php
$finder->in('ftp://example.com/pub/');
```

And it also works with user-defined streams:

Listing 23-6
```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";
}
```

3. http://api.symfony.com/2.1/Symfony/Component/Finder/SplFileInfo.html
4. http://php.net/manual/en/class.splfileinfo.php
5. http://www.php.net/manual/en/spl.iterators.php
6. http://php.net/manual/en/function.iterator-to-array.php
7. http://php.net/manual/en/function.iterator-count.php
8. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#in()
9. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#exclude()
10. http://www.php.net/manual/en/wrappers.php

Read the *Streams*[11] documentation to learn how to create your own streams.

Files or Directories

By default, the Finder returns files and directories; but the *files()*[12] and *directories()*[13] methods control that:

Listing 23-7
```
$finder->files();

$finder->directories();
```

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

Listing 23-8
```
$finder->files()->followLinks();
```

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

Listing 23-9
```
$finder->ignoreVCS(false);
```

Sorting

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

Listing 23-10
```
$finder->sortByName();

$finder->sortByType();
```

Notice that the `sort*` methods need to get all matching elements to do their jobs. For large iterators, it is slow.

You can also define your own sorting algorithm with `sort()` method:

Listing 23-11
```
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

File Name

Restrict files by name with the *name()*[14] method:

11. http://www.php.net/streams
12. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#files()
13. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#directories()
14. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#name()

Listing 23-12
```
$finder->files()->name('*.php');
```

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

Listing 23-13
```
$finder->files()->name('/\.php$/');
```

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

Listing 23-14
```
$finder->files()->notName('*.rb');
```

File Contents

New in version 2.1: Methods `contains()` and `notContains()` have been introduced in version 2.1.

Restrict files by contents with the *`contains()`*[15] method:

Listing 23-15
```
$finder->files()->contains('lorem ipsum');
```

The `contains()` method accepts strings or regexes:

Listing 23-16
```
$finder->files()->contains('/lorem\s+ipsum$/i');
```

The `notContains()` method excludes files containing given pattern:

Listing 23-17
```
$finder->files()->notContains('dolor sit amet');
```

File Size

Restrict files by size with the *`size()`*[16] method:

Listing 23-18
```
$finder->files()->size('< 1.5K');
```

Restrict by a size range by chaining calls:

Listing 23-19
```
$finder->files()->size('>= 1K')->size('<= 2K');
```

The comparison operator can be any of the following: `>`, `>=`, `<`, `<=`, `==`, `!=`.

New in version 2.1: The operator `!=` was added in version 2.1.

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*[17].

15. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#contains()
16. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#size()

File Date

Restrict files by last modified dates with the *`date()`*[18] method:

Listing 23-20
```
$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*[19] function.

Directory Depth

By default, the Finder recursively traverse directories. Restrict the depth of traversing with *`depth()`*[20]:

Listing 23-21
```
$finder->depth('== 0');
$finder->depth('< 3');
```

Custom Filtering

To restrict the matching file with your own strategy, use *`filter()`*[21]:

Listing 23-22
```
$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`*[22] instance. The file is excluded from the result set if the Closure returns `false`.

Reading contents of returned files

New in version 2.1: Method `getContents()` have been introduced in version 2.1.

The contents of returned files can be read with *`getContents()`*[23]:

Listing 23-23
```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);
```

17. http://physics.nist.gov/cuu/Units/binary.html
18. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#date()
19. http://www.php.net/manual/en/datetime.formats.php
20. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#depth()
21. http://api.symfony.com/2.1/Symfony/Component/Finder/Finder.html#filter()
22. http://api.symfony.com/2.1/Symfony/Component/Finder/SplFileInfo.html
23. http://api.symfony.com/2.1/Symfony/Component/Finder/SplFileInfo.html#getContents()

```
5
6 foreach ($finder as $file) {
7     $contents = $file->getContents();
8     ...
9 }
```

Chapter 24
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`, `$_FILE`, `$_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 PEAR (*pear.symfony.com/HttpFoundation*);
- Install it via Composer (*symfony/http-foundation* on Packagist).

Request

The most common way to create request is to base it on the current PHP global variables with *`createFromGlobals()`*[2]:

Listing 24-1
```php
use Symfony\Component\HttpFoundation\Request;

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

which is almost equivalent to the more verbose, but also more flexible, *`__construct()`*[3] call:

Listing 24-2

1. https://github.com/symfony/HttpFoundation
2. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#createFromGlobals()
3. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#__construct()

```
1  $request = new Request($_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 `$_FILE`;
- `server`: equivalent of `$_SERVER`;
- `headers`: mostly equivalent to a sub-set of `$_SERVER` (`$request->headers->get('Content-Type')`).

Each property is a *ParameterBag*[4] instance (or a sub-class of), which is a data holder class:

- `request`: *ParameterBag*[5];
- `query`: *ParameterBag*[6];
- `cookies`: *ParameterBag*[7];
- `attributes`: *ParameterBag*[8];
- `files`: *FileBag*[9];
- `server`: *ServerBag*[10];
- `headers`: *HeaderBag*[11].

All *ParameterBag*[12] instances have methods to retrieve and update its data:

- *all()*[13]: Returns the parameters;
- *keys()*[14]: Returns the parameter keys;
- *replace()*[15]: Replaces the current parameters by a new set;
- *add()*[16]: Adds parameters;
- *get()*[17]: Returns a parameter by name;
- *set()*[18]: Sets a parameter by name;
- *has()*[19]: Returns true if the parameter is defined;
- *remove()*[20]: Removes a parameter.

The *ParameterBag*[21] instance also has some methods to filter the input values:

---

4. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
5. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
6. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
7. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
8. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
9. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/FileBag.html
10. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ServerBag.html
11. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/HeaderBag.html
12. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
13. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#all()
14. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#keys()
15. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#replace()
16. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#add()
17. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#get()
18. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#set()
19. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#has()
20. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html#remove()
21. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html

- *getAlpha()*[22]: Returns the alphabetic characters of the parameter value;
- *getAlnum()*[23]: Returns the alphabetic characters and digits of the parameter value;
- *getDigits()*[24]: Returns the digits of the parameter value;
- *getInt()*[25]: Returns the parameter value converted to integer;
- *filter()*[26]: 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 24-3

```
// 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()*[27] via the third argument:

Listing 24-4

```
// 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'
```

Last, but not the least, you can also store additional data in the request, thanks to the `attributes` public property, which is also an instance of *ParameterBag*[28]. 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*.

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()*[29] method:

Listing 24-5

22. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getAlpha()
23. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getAlnum()
24. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getDigits()
25. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getInt()
26. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#filter()
27. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#get()
28. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ParameterBag.html
29. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getPathInfo()

```php
// for a request to http://example.com/blog/index.php/post/hello-world
// the path info is "/post/hello-world"
$request->getPathInfo();
```

Simulating a Request

Instead of creating a Request based on the PHP globals, you can also simulate a Request:

Listing 24-6
```php
$request = Request::create('/hello-world', 'GET', array('name' => 'Fabien'));
```

The *create()*[30] 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 an 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()*[31]:

Listing 24-7
```php
$request->overrideGlobals();
```

You can also duplicate an existing query via *duplicate()*[32] or change a bunch of parameters with a single call to *initialize()*[33].

Accessing the Session

If you have a session attached to the Request, you can access it via the *getSession()*[34] method; the *hasPreviousSession()*[35] method tells you if the request contains a Session which was started in one of the previous requests.

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*[36] 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 24-8
```php
use Symfony\Component\HttpFoundation\Response;

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

30. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#create()
31. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#overrideGlobals()
32. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#duplicate()
33. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#initialize()
34. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#getSession()
35. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html#hasPreviousSession()
36. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html

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

Listing 24-9
```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()`*[37] method:

Listing 24-10
```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()`*[38] method:

Listing 24-11
```php
$response->prepare($request);
```

Sending the response to the client is then as simple as calling *`send()`*[39]:

Listing 24-12
```php
$response->send();
```

Setting Cookies

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

Listing 24-13
```php
use Symfony\Component\HttpFoundation\Cookie;

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

The *`setCookie()`*[40] method takes an instance of *`Cookie`*[41] as an argument.

You can clear a cookie via the *`clearCookie()`*[42] method.

Managing the HTTP Cache

The *`Response`*[43] class has a rich set of methods to manipulate the HTTP headers related to the cache:

- *`setPublic()`*[44];
- *`setPrivate()`*[45];
- *`expire()`*[46];

37. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setCharset()
38. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#prepare()
39. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#send()
40. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/ResponseHeaderBag.html#setCookie()
41. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Cookie.html
42. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#clearCookie()
43. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html
44. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setPublic()
45. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setPrivate()

- *setExpires()*[47];
- *setMaxAge()*[48];
- *setSharedMaxAge()*[49];
- *setTtl()*[50];
- *setClientTtl()*[51];
- *setLastModified()*[52];
- *setEtag()*[53];
- *setVary()*[54];

The *setCache()*[55] method can be used to set the most commonly used cache information in one method call:

Listing 24-14
```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()*[56] method:

Listing 24-15
```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*[57] class:

Listing 24-16
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

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

46. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#expire()
47. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setExpires()
48. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setMaxAge()
49. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setSharedMaxAge()
50. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setTtl()
51. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setClientTtl()
52. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setLastModified()
53. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setEtag()
54. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setVary()
55. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#setCache()
56. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#isNotModified()
57. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/RedirectResponse.html

Streaming a Response

New in version 2.1: Support for streamed responses was added in Symfony 2.1.

The *`StreamedResponse`*[58] class allows you to stream the Response back to the client. The response content is represented by a PHP callable instead of a string:

Listing 24-17
```php
use Symfony\Component\HttpFoundation\StreamedResponse;

$response = new StreamedResponse();
$response->setCallback(function () {
    echo 'Hello World';
    flush();
    sleep(2);
    echo 'Hello World';
    flush();
});
$response->send();
```

Downloading Files

New in version 2.1: The `makeDisposition` method was added in Symfony 2.1.

When uploading a file, you must add a `Content-Disposition` header to your response. While creating this header for basic file downloads is easy, using non-ASCII filenames is more involving. The *`makeDisposition()`*[59] abstracts the hard work behind a simple API:

Listing 24-18
```php
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

$d = $response->headers->makeDisposition(ResponseHeaderBag::DISPOSITION_ATTACHMENT,
'foo.pdf');

$response->headers->set('Content-Disposition', $d);
```

Session

The session information is in its own document: *Session Management*.

58. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/StreamedResponse.html
59. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Response.html#makeDisposition()

Chapter 25
Session Management

The Symfony2 HttpFoundation Component has a very powerful and flexible session subsystem which is designed to provide session management through a simple object-oriented interface using a variety of session storage drivers.

New in version 2.1: The *SessionInterface*[1] interface, as well as a number of other changes, are new as of Symfony 2.1.

Sessions are used via the simple *Session*[2] implementation of *SessionInterface*[3] interface.

Quick example:

Listing 25-1

```
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
$session->start();

// set and get session attributes
$session->set('name', 'Drak');
$session->get('name');

// set flash messages
$session->getFlashBag()->add('notice', 'Profile updated');

// retrieve messages
foreach ($session->getFlashBag()->get('notice', array()) as $message) {
    echo "<div class='flash-notice'>$message</div>";
}
```

1. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionInterface.html
2. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html
3. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionInterface.html

Symfony sessions are designed to replace several native PHP funtions. Applications should avoid using `session_start()`, `session_regenerate_id()`, `session_id()`, `session_name()`, and `session_destroy()` and instead use the APIs in the following section.

While it is recommended to explicitly start a session, a sessions will actually start on demand, that is, if any session request is made to read/write session data.

Symfony sessions are incompatible with PHP ini directive `session.auto_start = 1` This directive should be turned off in `php.ini`, in the webserver directives or in `.htaccess`.

Session API

The *`Session`*[4] class implements *`SessionInterface`*[5].

The *`Session`*[6] has a simple API as follows divided into a couple of groups.

Session workflow

- *`start()`*[7]: Starts the session - do not use `session_start()`.
- *`migrate()`*[8]: Regenerates the session ID - do not use `session_regenerate_id()`. This method can optionally change the lifetime of the new cookie that will be emitted by calling this method.
- *`invalidate()`*[9]: Clears all session data and regenerates session ID. Do not use `session_destroy()`.
- *`getId()`*[10]: Gets the session ID. Do not use `session_id()`.
- *`setId()`*[11]: Sets the session ID. Do not use `session_id()`.
- *`getName()`*[12]: Gets the session name. Do not use `session_name()`.
- *`setName()`*[13]: Sets the session name. Do not use `session_name()`.

Session attributes

- *`set()`*[14]: Sets an attribute by key;
- *`get()`*[15]: Gets an attribute by key;
- *`all()`*[16]: Gets all attributes as an array of key => value;
- *`has()`*[17]: Returns true if the attribute exists;
- *`keys()`*[18]: Returns an array of stored attribute keys;

---

4. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html
5. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionInterface.html
6. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html
7. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#start()
8. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#migrate()
9. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#invalidate()
10. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getId()
11. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#setId()
12. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getName()
13. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#setName()
14. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#set()
15. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#get()
16. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#all()
17. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#has()
18. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#keys()

- *replace()*[19]: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.
- *remove()*[20]: Deletes an attribute by key;
- *clear()*[21]: Clear all attributes;

The attributes are stored internally in an "Bag", a PHP object that acts like an array. A few methods exist for "Bag" management:

- *registerBag()*[22]: Registers a *SessionBagInterface*[23]
- *getBag()*[24]: Gets a *SessionBagInterface*[25] by bag name.
- *getFlashBag()*[26]: Gets the *FlashBagInterface*[27]. This is just a shortcut for convenience.

Session meta-data

- *getMetadataBag()*[28]: Gets the *StorageMetadataBag*[29] which contains information about the session.

Session Data Management

PHP's session management requires the use of the `$_SESSION` super-global, however, this interferes somewhat with code testability and encapsulation in a OOP paradigm. To help overcome this, Symfony2 uses 'session bags' linked to the session to encapsulate a specific dataset of 'attributes' or 'flash messages'.

This approach also mitigates namespace pollution within the `$_SESSION` super-global because each bag stores all its data under a unique namespace. This allows Symfony2 to peacefully co-exist with other applications or libraries that might use the `$_SESSION` super-global and all data remains completely compatible with Symfony2's session management.

Symfony2 provides 2 kinds of storage bags, with two separate implementations. Everything is written against interfaces so you may extend or create your own bag types if necessary.

SessionBagInterface[30] has the following API which is intended mainly for internal purposes:

- *getStorageKey()*[31]: Returns the key which the bag will ultimately store its array under in `$_SESSION`. Generally this value can be left at its default and is for internal use.
- *initialize()*[32]: This is called internally by Symfony2 session storage classes to link bag data to the session.
- *getName()*[33]: Returns the name of the session bag.

19. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#replace()
20. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#remove()
21. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#clear()
22. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#registerBag()
23. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
24. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getBag()
25. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
26. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getFlashBag()
27. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
28. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getMetadataBag()
29. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/StorageMetadataBag.html
30. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html
31. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#getStorageKey()
32. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#initialize()
33. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionBagInterface.html#getName()

Attributes

The purpose of the bags implementing the *AttributeBagInterface*[34] is to handle session attribute storage. This might include things like user ID, and remember me login settings or other user based state information.

- *AttributeBag*[35] This is the standard default implementation.
- *NamespacedAttributeBag*[36] This implementation allows for attributes to be stored in a structured namespace.

Any plain *key* => *value* storage system is limited in the extent to which complex data can be stored since each key must be unique. You can achieve namespacing by introducing a naming convention to the keys so different parts of your application could operate without clashing. For example, *module1.foo* and *module2.foo*. However, sometimes this is not very practical when the attributes data is an array, for example a set of tokens. In this case, managing the array becomes a burden because you have to retrieve the array then process it and store it again:

Listing 25-2
```
1 $tokens = array('tokens' => array('a' => 'a6c1e0b6',
2                                   'b' => 'f4a7b1f3'));
```

So any processing of this might quickly get ugly, even simply adding a token to the array:

Listing 25-3
```
1 $tokens = $session->get('tokens');
2 $tokens['c'] = $value;
3 $session->set('tokens', $tokens);
```

With structured namespacing, the key can be translated to the array structure like this using a namespace character (defaults to */*):

Listing 25-4
```
1 $session->set('tokens/c', $value);
```

This way you can easily access a key within the stored array directly and easily.

AttributeBagInterface[37] has a simple API

- *set()*[38]: Sets an attribute by key;
- *get()*[39]: Gets an attribute by key;
- *all()*[40]: Gets all attributes as an array of key => value;
- *has()*[41]: Returns true if the attribute exists;
- *keys()*[42]: Returns an array of stored attribute keys;
- *replace()*[43]: Sets multiple attributes at once: takes a keyed array and sets each key => value pair.
- *remove()*[44]: Deletes an attribute by key;
- *clear()*[45]: Clear the bag;

34. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
35. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBag.html
36. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/NamespacedAttributeBag.html
37. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html
38. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#set()
39. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#get()
40. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#all()
41. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#has()
42. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#keys()
43. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#replace()
44. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#remove()
45. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Attribute/AttributeBagInterface.html#clear()

Flash messages

The purpose of the *FlashBagInterface*[46] is to provide a way of settings and retrieving messages on a per session basis. The usual workflow for flash messages would be set in an request, and displayed after a page redirect. For example, a user submits a form which hits an update controller, and after processing the controller redirects the page to either the updated page or an error page. Flash messages set in the previous page request would be displayed immediately on the subsequent page load for that session. This is however just one application for flash messages.

- *AutoExpireFlashBag*[47]

  This implementation messages set in one page-load will be available for display only on the next page load. These messages will auto expire regardless of if they are retrieved or not.

- *FlashBag*[48]

  In this implementation, messages will remain in the session until they are explicitly retrieved or cleared. This makes it possible to use ESI caching.

FlashBagInterface[49] has a simple API

- *add()*[50]: Adds a flash message to the stack of specified type;
- *set()*[51]: Sets flashes by type; This method conveniently takes both singles messages as a `string` or multiple messages in an `array`.
- *get()*[52]: Gets flashes by type and clears those flashes from the bag;
- *setAll()*[53]: Sets all flashes, accepts a keyed array of arrays `type => array(messages)`;
- *all()*[54]: Gets all flashes (as a keyed array of arrays) and clears the flashes from the bag;
- *peek()*[55]: Gets flashes by type (read only);
- *peekAll()*[56]: Gets all flashes (read only) as keyed array of arrays;
- *has()*[57]: Returns true if the type exists, false if not;
- *keys()*[58]: Returns an array of the stored flash types;
- *clear()*[59]: Clears the bag;

For simple applications it is usually sufficient to have one flash message per type, for example a confirmation notice after a form is submitted. However, flash messages are stored in a keyed array by flash `$type` which means your application can issue multiple messages for a given type. This allows the API to be used for more complex messaging in your application.

Examples of setting multiple flashes:

Listing 25-5
```php
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session();
```

46. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
47. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/AutoExpireFlashBag.html
48. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBag.html
49. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html
50. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#add()
51. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#set()
52. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#get()
53. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#setAll()
54. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#all()
55. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#peek()
56. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#peekAll()
57. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#has()
58. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#keys()
59. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#clear()

```
4  $session->start();
5
6  // add flash messages
7  $session->getFlashBag()->add('warning', 'Your config file is writable, it should be set
8  read-only');
9  $session->getFlashBag()->add('error', 'Failed to update name');
   $session->getFlashBag()->add('error', 'Another error');
```

Displaying the flash messages might look like this:

Simple, display one type of message:

Listing 25-6
```
1  // display warnings
2  foreach ($session->getFlashBag()->get('warning', array()) as $message) {
3      echo "<div class='flash-warning'>$message</div>";
4  }
5
6  // display errors
7  foreach ($session->getFlashBag()->get('error', array()) as $message) {
8      echo "<div class='flash-error'>$message</div>";
9  }
```

Compact method to process display all flashes at once:

Listing 25-7
```
1  foreach ($session->getFlashBag()->all() as $type => $messages) {
2      foreach ($messages as $message) {
3          echo "<div class='flash-$type'>$message</div>\n";
4      }
5  }
```

Chapter 26
Configuring Sessions and Save Handlers

This section deals with how to configure session management and fine tune it to your specific needs. This documentation covers save handlers, which store and retrieve session data, and configuring session behaviour.

Save Handlers

The PHP session workflow has 6 possible operations that may occur. The normal session follows *open*, *read*, *write* and *close*, with the possibility of *destroy* and *gc* (garbage collection which will expire any old sessions: *gc* is called randomly according to PHP's configuration and if called, it is invoked after the *open* operation). You can read more about this at *php.net/session.customhandler*[1]

Native PHP Save Handlers

So-called 'native' handlers, are save handlers which are either compiled into PHP or provided by PHP extensions, such as PHP-Sqlite, PHP-Memcached and so on.

All native save handlers are internal to PHP and as such, have no public facing API. They must be configured by PHP ini directives, usually `session.save_path` and potentially other driver specific directives. Specific details can be found in docblock of the `setOptions()` method of each class.

While native save handlers can be activated by directly using `ini_set('session.save_handler', $name);`, Symfony2 provides a convenient way to activate these in the same way as custom handlers.

Symfony2 provides drivers for the following native save handler as an example:

- *`NativeFileSessionHandler`*[2]

Example usage:

Listing 26-1
```
1 use Symfony\Component\HttpFoundation\Session\Session;
2 use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
```

1. http://php.net/session.customhandler
2. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeFileSessionHandler.html

```php
use Symfony\Component\HttpFoundation\Session\Storage\Handler\NativeFileSessionHandler;

$storage = new NativeSessionStorage(array(), new NativeFileSessionHandler());
$session = new Session($storage);
```

With the exception of the `files` handler which is built into PHP and always available, the availability of the other handlers depends on those PHP extensions being active at runtime.

Native save handlers provide a quick solution to session storage, however, in complex systems where you need more control, custom save handlers may provide more freedom and flexibility. Symfony2 provides several implementations which you may further customise as required.

Custom Save Handlers

Custom handlers are those which completely replace PHP's built in session save handlers by providing six callback functions which PHP calls internally at various points in the session workflow.

Symfony2 HttpFoundation provides some by default and these can easily serve as examples if you wish to write your own.

- *PdoSessionHandler*[3]
- *MemcacheSessionHandler*[4]
- *MemcachedSessionHandler*[5]
- *NullSessionHandler*[6]

Example usage:

Listing 26-2
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\SessionStorage;
use Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler;

$storage = new NativeSessionStorage(array(), new PdoSessionHandler());
$session = new Session($storage);
```

Configuring PHP Sessions

The *NativeSessionStorage*[7] can configure most of the PHP ini configuration directives which are documented at *php.net/session.configuration*[8].

To configure these setting, pass the keys (omitting the initial `session.` part of the key) as a key-value array to the `$options` constructor argument. Or set them via the *setOptions()*[9] method.

For the sake of clarity, some key options are explained in this documentation.

3. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html
4. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcacheSessionHandler.html
5. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/MemcachedSessionHandler.html
6. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/NullSessionHandler.html
7. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
8. http://php.net/session.configuration
9. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#setOptions()

Session Cookie Lifetime

For security, session tokens are generally recommended to be sent as session cookies. You can configure the lifetime of session cookies by specifying the lifetime (in seconds) using the `cookie_lifetime` key in the constructor's `$options` argument in *`NativeSessionStorage`*[10].

Setting a `cookie_lifetime` to `0` will cause the cookie to live only as long as the browser remains open. Generally, `cookie_lifetime` would be set to a relatively large number of days, weeks or months. It is not uncommon to set cookies for a year or more depending on the application.

Since session cookies are just a client-side token, they are less important in controlling the fine details of your security settings which ultimately can only be securely controlled from the server side.

The `cookie_lifetime` setting is the number of seconds the cookie should live for, it is not a Unix timestamp. The resulting session cookie will be stamped with an expiry time of `time()``+``cookie_lifetime` where the time is taken from the server.

Configuring Garbage Collection

When a session opens, PHP will call the `gc` handler randomly according to the probability set by `session.gc_probability` / `session.gc_divisor`. For example if these were set to `5/100` respectively, it would mean a probability of 5%. Similarly, `3/4` would mean a 3 in 4 chance of being called, i.e. 75%.

If the garbage collection handler is invoked, PHP will pass the value stored in the PHP ini directive `session.gc_maxlifetime`. The meaning in this context is that any stored session that was saved more than ``maxlifetime` ago should be deleted. This allows one to expire records based on idle time.

You can configure these settings by passing `gc_probability`, `gc_divisor` and `gc_maxlifetime` in an array to the constructor of *`NativeSessionStorage`*[11] or to the *`setOptions()`*[12] method.

Session Lifetime

When a new session is created, meaning Symfony2 issues a new session cookie to the client, the cookie will be stamped with an expiry time. This is calculated by adding the PHP runtime configuration value in `session.cookie_lifetime` with the current server time.

PHP will only issue a cookie once. The client is expected to store that cookie for the entire lifetime. A new cookie will only be issued when the session is destroyed, the browser cookie is deleted, or the session ID is regenerated using the `migrate()` or `invalidate()` methods of the `Session` class.

The initial cookie lifetime can be set by configuring `NativeSessionStorage` using the `setOptions(array('cookie_lifetime' => 1234))` method.

A cookie lifetime of `0` means the cookie expire when the browser is closed.

10. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
11. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
12. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html#setOptions()

Session Idle Time/Keep Alive

There are often circumstances where you may want to protect, or minimize unauthorized use of a session when a user steps away from their terminal while logged in by destroying the session after a certain period of idle time. For example, it is common for banking applications to log the user out after just 5 to 10 minutes of inactivity. Setting the cookie lifetime here is not appropriate because that can be manipulated by the client, so we must do the expiry on the server side. The easiest way is to implement this via garbage collection which runs reasonably frequently. The cookie `lifetime` would be set to a relatively high value, and the garbage collection `maxlifetime` would be set to destroy sessions at whatever the desired idle period is.

The other option is to specifically checking if a session has expired after the session is started. The session can be destroyed as required. This method of processing can allow the expiry of sessions to be integrated into the user experience, for example, by displaying a message.

Symfony2 records some basic meta-data about each session to give you complete freedom in this area.

Session meta-data

Sessions are decorated with some basic meta-data to enable fine control over the security settings. The session object has a getter for the meta-data, *`getMetadataBag()`*[13] which exposes an instance of *`MetadataBag`*[14]:

Listing 26-3
```
$session->getMetadataBag()->getCreated();
$session->getMetadataBag()->getLastUsed();
```

Both methods return a Unix timestamp (relative to the server).

This meta-data can be used to explicitly expire a session on access, e.g.:

Listing 26-4
```
$session->start();
if (time() - $session->getMetadataBag()->getLastUpdate() > $maxIdleTime) {
    $session->invalidate();
    throw new SessionExpired(); // redirect to expired session page
}
```

It is also possible to tell what the `cookie_lifetime` was set to for a particular cookie by reading the `getLifetime()` method:

Listing 26-5
```
$session->getMetadataBag()->getLifetime();
```

The expiry time of the cookie can be determined by adding the created timestamp and the lifetime.

PHP 5.4 compatibility

Since PHP 5.4.0, *`SessionHandler`*[15] and *`SessionHandlerInterface`*[16] are available. Symfony 2.1 provides forward compatibility for the *`SessionHandlerInterface`*[17] so it can be used under PHP 5.3. This greatly improves inter-operability with other libraries.

13. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Session.html#getMetadataBag()
14. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/MetadataBag.html
15. http://php.net/manual/en/class.sessionhandler.php
16. http://php.net/manual/en/class.sessionhandlerinterface.php
17. http://php.net/manual/en/class.sessionhandlerinterface.php

SessionHandler[18] is a special PHP internal class which exposes native save handlers to PHP user-space.

In order to provide a solution for those using PHP 5.4, Symfony2 has a special class called *NativeSessionHandler*[19] which under PHP 5.4, extends from *SessionHandler* and under PHP 5.3 is just a empty base class. This provides some interesting opportunities to leverage PHP 5.4 functionality if it is available.

Save Handler Proxy

There are two kinds of save handler class proxies which inherit from *AbstractProxy*[20]: they are *NativeProxy*[21] and *SessionHandlerProxy*[22].

NativeSessionStorage[23] automatically injects storage handlers into a save handler proxy unless already wrapped by one.

NativeProxy[24] is used automatically under PHP 5.3 when internal PHP save handlers are specified using the *Native*SessionHandler* classes, while *SessionHandlerProxy*[25] will be used to wrap any custom save handlers, that implement *SessionHandlerInterface*[26].

Under PHP 5.4 and above, all session handlers implement *SessionHandlerInterface*[27] including *Native*SessionHandler* classes which inherit from *SessionHandler*[28].

The proxy mechanism allow you to get more deeply involved in session save handler classes. A proxy for example could be used to encrypt any session transaction without knowledge of the specific save handler.

18. http://php.net/manual/en/class.sessionhandler.php
19. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeSessionHandler.html
20. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/AbstractProxy.html
21. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeProxy.html
22. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/SessionHandlerProxy.html
23. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
24. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeProxy.html
25. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/Handler/SessionHandlerProxy.html
26. http://php.net/manual/en/class.sessionhandlerinterface.php
27. http://php.net/manual/en/class.sessionhandlerinterface.php
28. http://php.net/manual/en/class.sessionhandler.php

Chapter 27
Testing with Sessions

Symfony2 is designed from the ground up with code-testability in mind. In order to make your code which utilizes session easily testable we provide two separate mock storage mechanisms for both unit testing and functional testing.

Testing code using real sessions is tricky because PHP's workflow state is global and it is not possible to have multiple concurrent sessions in the same PHP process.

The mock storage engines simulate the PHP session workflow without actually starting one allowing you to test your code without complications. You may also run multiple instances in the same PHP process.

The mock storage drivers do not read or write the system globals *session_id()* or *session_name()*. Methods are provided to simulate this if required:

- *getId()*[1]: Gets the session ID.
- *setId()*[2]: Sets the session ID.
- *getName()*[3]: Gets the session name.
- *setName()*[4]: Sets the session name.

Unit Testing

For unit testing where it is not necessary to persist the session, you should simply swap out the default storage engine with *MockArraySessionStorage*[5]:

Listing 27-1
```php
use Symfony\Component\HttpFoundation\Session\Storage\MockArraySessionStorage;
use Symfony\Component\HttpFoundation\Session\Session;

$session = new Session(new MockArraySessionStorage());
```

1. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#getId()
2. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#setId()
3. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#getName()
4. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/SessionStorageInterface.html#setName()
5. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/MockArraySessionStorage.html

Functional Testing

For functional testing where you may need to persist session data across separate PHP processes, simply change the storage engine to *MockFileSessionStorage*[6]:

Listing 27-2
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\MockFileSessionStorage;

$session = new Session(new MockFileSessionStorage());
```

6. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Session/Storage/MockFileSessionStorage.html

Chapter 28
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 PEAR (*pear.symfony.com/Locale*);
- Install it via Composer (*symfony/locale* on Packagist).

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 28-1
```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[10] class enriches native *Locale*[11] class with additional features:

Listing 28-2
```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();

// Get ICU versions
$icuVersion = Locale::getIcuVersion();
$icuDataVersion = Locale::getIcuDataVersion();
```

10. http://api.symfony.com/2.1/Symfony/Component/Locale/Locale.html
11. http://php.net/manual/en/class.locale.php

Chapter 29
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 PEAR (*pear.symfony.com/Process*);
- Install it via Composer (*symfony/process* on Packagist).

Usage

The *Process*[2] class allows you to execute a command in a sub-process:

Listing 29-1
```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()*[3] method takes care of the subtle differences between the different platforms when executing the command.

1. https://github.com/symfony/Process
2. http://api.symfony.com/2.1/Symfony/Component/Process/Process.html
3. http://api.symfony.com/2.1/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()*[4] method:

Listing 29-2
```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 29-3
```php
use Symfony\Component\Process\PhpProcess;

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

New in version 2.1: The `ProcessBuilder` class has been as of 2.1.

To make your code work better on all platforms, you might want to use the *ProcessBuilder*[5] class instead:

Listing 29-4
```php
use Symfony\Component\Process\ProcessBuilder;

$builder = new ProcessBuilder(array('ls', '-lsa'));
$builder->getProcess()->run();
```

4. http://api.symfony.com/2.1/Symfony/Component/Process/Process.html#run()
5. http://api.symfony.com/2.1/Symfony/Component/Process/ProcessBuilder.html

Chapter 30
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 PEAR (*pear.symfony.com/Routing*);
- Install it via Composer (*symfony/routing* on Packagist)

Usage

In order to set up a basic routing system you need three parts:

- A *`RouteCollection`*[2], which contains the route definitions (instances of the class *`Route`*[3])
- A *`RequestContext`*[4], which has information about the request
- A *`UrlMatcher`*[5], 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 30-1

```
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

```

1. https://github.com/symfony/Routing
2. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html
3. http://api.symfony.com/2.1/Symfony/Component/Routing/Route.html
4. http://api.symfony.com/2.1/Symfony/Component/Routing/RequestContext.html
5. http://api.symfony.com/2.1/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*[6].

The *RouteCollection::add()*[7] method takes two arguments. The first is the name of the route. The second is a *Route*[8] 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*[9] 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 30-2

```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');
```

6. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html
7. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html#add()
8. http://api.symfony.com/2.1/Symfony/Component/Routing/Route.html
9. http://api.symfony.com/2.1/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 30-3
```
$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 30-4
```
$route = new Route(
    '/start/{suffix}',
    array('suffix' => ''),
    array('suffix' => '.*')
);
```

Using Prefixes

You can add routes or other instances of *`RouteCollection`*[10] 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 30-5
```
$rootCollection = new RouteCollection();

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

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

10. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html

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

Set the Request Parameters

The *RequestContext*[11] provides information about the current request. You can define all parameters of an HTTP request with this class via its constructor:

Listing 30-6

```
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*[12]. But If you use the *HttpFoundation* component, you can use its *Request*[13] class to feed the *RequestContext*[14] in a shortcut:

Listing 30-7

```
1  use Symfony\Component\HttpFoundation\Request;
2
3  $context = new RequestContext();
4  $context->fromRequest(Request::createFromGlobals());
```

Generate a URL

While the *UrlMatcher*[15] tries to find a route that fits the given request you can also build a URL from a certain route:

Listing 30-8

```
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
```

11. http://api.symfony.com/2.1/Symfony/Component/Routing/RequestContext.html
12. http://api.symfony.com/2.1/Symfony/Component/Routing/RequestContext.html
13. http://api.symfony.com/2.1/Symfony/Component/HttpFoundation/Request.html
14. http://api.symfony.com/2.1/Symfony/Component/Routing/RequestContext.html
15. http://api.symfony.com/2.1/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*[16] 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*[17] instance as the constructor argument. You can use the *FileLocator*[18] 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*[19].

If you're using the `YamlFileLoader`, then route definitions look like this:

Listing 30-9
```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 30-10
```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*[20] there are two other loaders that work the same way:

- *XmlFileLoader*[21]
- *PhpFileLoader*[22]

If you use the *PhpFileLoader*[23] you have to provide the name of a php file which returns a *RouteCollection*[24]:

Listing 30-11
```php
// RouteProvider.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
```

16. http://api.symfony.com/2.1/Symfony/Component/Routing/RequestContext.html
17. http://api.symfony.com/2.1/Symfony/Component/Config/FileLocator.html
18. http://api.symfony.com/2.1/Symfony/Component/Config/FileLocator.html
19. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html
20. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/YamlFileLoader.html
21. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/XmlFileLoader.html
22. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/PhpFileLoader.html
23. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/PhpFileLoader.html
24. http://api.symfony.com/2.1/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*[25], which calls a closure and uses the result as a *RouteCollection*[26]:

Listing 30-12
```
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*[27] and *AnnotationFileLoader*[28] to load route definitions from class annotations. The specific details are left out here.

The all-in-one Router

The *Router*[29] 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 30-13
```
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*[30] class would look like:

Listing 30-14
```
1  $locator = new FileLocator(array(__DIR__));
2  $requestContext = new RequestContext($_SERVER['REQUEST_URI']);
3
```

25. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/ClosureLoader.html
26. http://api.symfony.com/2.1/Symfony/Component/Routing/RouteCollection.html
27. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/AnnotationDirectoryLoader.html
28. http://api.symfony.com/2.1/Symfony/Component/Routing/Loader/AnnotationFileLoader.html
29. http://api.symfony.com/2.1/Symfony/Component/Routing/Router.html
30. http://api.symfony.com/2.1/Symfony/Component/Routing/Router.html

```php
$router = new Router(
    new YamlFileLoader($locator),
    "routes.yml",
    array('cache_dir' => __DIR__.'/cache'),
    $requestContext,
);
$router->match('/foo/bar');
```

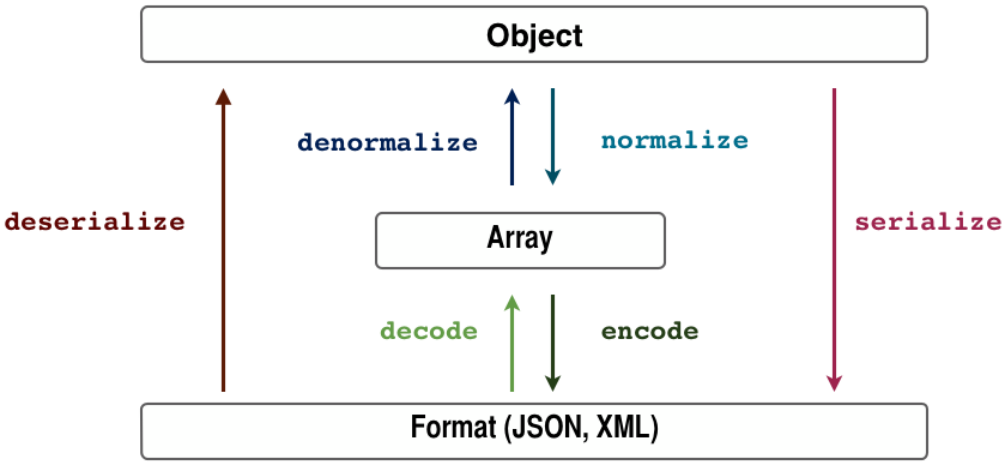
If you use caching, the Routing component will compile new classes which are saved in the `cache_dir`. This means your script must have write permissions for that location.

Chapter 31

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 PEAR (*pear.symfony.com/Serializer*);

- Install it via Composer (*symfony/serializer* on Packagist).

Usage

Using the Serializer component is really simple. You just need to set up the *Serializer*[2] specifying which Encoders and Normalizer are going to be available:

Listing 31-1
```php
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;

$encoders = array(new XmlEncoder(), new JsonEncoder());
$normalizers = array(new GetSetMethodNormalizer());

$serializer = new Serializer($normalizers, $encoders);
```

Serializing an object

For the sake of this example, let's assume the following class already exists in our project:

Listing 31-2
```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:

1. https://github.com/symfony/Serializer
2. http://api.symfony.com/2.1/Symfony/Component/Serializer/Serializer.html

Listing 31-3
```php
$person = new Acme\Person();
$person->setName('foo');
$person->setAge(99);

$serializer->serialize($person, 'json'); // Output: {"name":"foo","age":99}
```

The first parameter of the *serialize()*[3] is the object to be serialized and the second is used to choose the proper encoder, in this case *JsonEncoder*[4].

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 31-4
```php
$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
</person>
EOF;

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

In this case, *deserialize()*[5] 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

JMSSerializationBundle

A popular third-party bundle, *JMSSerializationBundle*[6] exists and extends (and sometimes replaces) the serialization functionality. This 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).

3. http://api.symfony.com/2.1/Symfony/Component/Serializer/Serializer.html#serialize()
4. http://api.symfony.com/2.1/Symfony/Component/Serializer/Encoder/JsonEncoder.html
5. http://api.symfony.com/2.1/Symfony/Component/Serializer/Serializer.html#deserialize()
6. https://github.com/schmittjoh/JMSSerializerBundle

Chapter 32
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 PEAR (*pear.symfony.com/Templating*);
- Install it via Composer (*symfony/templating* on Packagist).

Usage

The *PhpEngine*[2] class is the entry point of the component. It needs a template name parser (*TemplateNameParserInterface*[3]) to convert a template name to a template reference and template loader (*LoaderInterface*[4]) to find the template associated to a reference:

Listing 32-1
```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. http://api.symfony.com/2.1/Symfony/Component/Templating/PhpEngine.html
3. http://api.symfony.com/2.1/Symfony/Component/Templating/TemplateNameParserInterface.html
4. http://api.symfony.com/2.1/Symfony/Component/Templating/Loader/LoaderInterface.html

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

The *render()*[5] method executes the file *views/hello.php* and returns the output text.

Listing 32-2
```
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 32-3
```
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()*[6] method is called in the sub-template to set its parent template.

Listing 32-4
```
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*[7] helper must be registered:

Listing 32-5
```
1  use Symfony\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));
```

5. http://api.symfony.com/2.1/Symfony/Component/Templating/PhpEngine.html#render()
6. http://api.symfony.com/2.1/Symfony/Component/Templating/PhpEngine.html#extend()
7. http://api.symfony.com/2.1/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 33
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.

Installation

You can install the component in many different ways:

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

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.

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

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*[3]), and the other dumps a PHP array to a YAML string (*Dumper*[4]).

On top of these two classes, the *Yaml*[5] class acts as a thin wrapper that simplifies common uses.

Reading YAML Files

The *parse()*[6] method parses a YAML string and converts it to a PHP array:

Listing 33-1
```php
use Symfony\Component\Yaml\Parser;

$yaml = new Parser();

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

If an error occurs during parsing, the parser throws a *ParseException*[7] exception indicating the error type and the line in the original YAML string where the error occurred:

Listing 33-2
```php
use Symfony\Component\Yaml\Exception\ParseException;

try {
```

3. http://api.symfony.com/2.1/Symfony/Component/Yaml/Parser.html
4. http://api.symfony.com/2.1/Symfony/Component/Yaml/Dumper.html
5. http://api.symfony.com/2.1/Symfony/Component/Yaml/Yaml.html
6. http://api.symfony.com/2.1/Symfony/Component/Yaml/Parser.html#parse()
7. http://api.symfony.com/2.1/Symfony/Component/Yaml/Exception/ParseException.html

```
    $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()*[8] wrapper method:

Listing 33-3
```
use Symfony\Component\Yaml\Yaml;

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

The *parse()*[9] static method takes a YAML string or a file containing YAML. Internally, it calls the *parse()*[10] method, but enhances the error if something goes wrong by adding the filename to the message.

Executing PHP Inside YAML Files

New in version 2.1: The `Yaml::enablePhpParsing()` method is new to Symfony 2.1. Prior to 2.1, PHP was *always* executed when calling the `parse()` function.

By default, if you include PHP inside a YAML file, it will not be parsed. If you do want PHP to be parsed, you must call `Yaml::enablePhpParsing()` before parsing the file to activate this mode. If you only want to allow PHP code for a single YAML file, be sure to disable PHP parsing after parsing the single file by calling `Yaml::$enablePhpParsing = false;` (`$enablePhpParsing` is a public property).

Writing YAML Files

The *dump()*[11] method dumps any PHP array to its YAML representation:

Listing 33-4
```
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/2.1/Symfony/Component/Yaml/Yaml.html#parse()
9. http://api.symfony.com/2.1/Symfony/Component/Yaml/Yaml.html#parse()
10. http://api.symfony.com/2.1/Symfony/Component/Yaml/Parser.html#parse()
11. http://api.symfony.com/2.1/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*[12] exception.

If you only need to dump one array, you can use the *dump()*[13] static method shortcut:

Listing 33-5
```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 33-6
```
{ foo: bar, bar: { foo: bar, bar: baz } }
```

The second argument of the *dump()*[14] method customizes the level at which the output switches from the expanded representation to the inline one:

Listing 33-7
```php
echo $dumper->dump($array, 1);
```

Listing 33-8
```
foo: bar
bar: { foo: bar, bar: baz }
```

Listing 33-9
```php
echo $dumper->dump($array, 2);
```

Listing 33-10
```
foo: bar
bar:
    foo: bar
    bar: baz
```

The YAML Format

According to the official *YAML*[15] 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).

12. http://api.symfony.com/2.1/Symfony/Component/Yaml/Exception/DumpException.html
13. http://api.symfony.com/2.1/Symfony/Component/Yaml/Yaml.html#dump()
14. http://api.symfony.com/2.1/Symfony/Component/Yaml/Dumper.html#dump()
15. http://yaml.org/

Scalars

The syntax for scalars is similar to the PHP syntax.

Strings

Listing 33-11
```
A string in YAML
```

Listing 33-12
```
'A singled-quoted string in YAML'
```

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

Listing 33-13
```
'A single quote '' in a single-quoted string'
```

Listing 33-14
```
"A double-quoted string in YAML\n"
```

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 33-15
```
|
  \/ /| |\/| |
  / / | |  | |__
```

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

Listing 33-16
```
>
  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 33-17
```
# an integer
12
```

Listing 33-18
```
# an octal
014
```

Listing 33-19
```
# an hexadecimal
0xC
```

Listing 33-20
```
# a float
13.4
```

Listing 33-21
```
# an exponential number
1.2e+34
```

Listing 33-22
```
# 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 33-23
```
2001-12-14t21:59:43.10-05:00
```

Listing 33-24
```
# 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 33-25
```
- PHP
- Perl
- Python
```

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

Listing 33-26
```
array('PHP', 'Perl', 'Python');
```

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

Listing 33-27
```
PHP: 5.2
MySQL: 5.1
Apache: 2.2.20
```

which is equivalent to this PHP code:

Listing 33-28
```
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 33-29
```
PHP:    5.2
MySQL:  5.1
Apache: 2.2.20
```

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

Listing 33-30
```
"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 33-31
```
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 33-32
```
'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 33-33
```
[PHP, Perl, Python]
```

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

Listing 33-34
```
{ PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

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

Listing 33-35
```
'Chapter 1': [Introduction, Event Types]
'Chapter 2': [Introduction, Helpers]
```

Listing 33-36
```
"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 33-37
```
# 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.