SOMMAIRE

1. Standard de codage

1.1 Introduction

Symfony[1] suit les standards définis dans les documents PSR-0, PSR-1, PSR-2 and PSR-4.

Ces standards sont généralisables à tout langages dans certaine mesure, c'est pourquoi nous allons utiliser ces standards de codage pour nos deux applications afin que la compréhension et la qualité du code soit la meilleure.

Les documents définissant les standards étant totalement en anglais et sachant que ces documents seront seulement utilisés par les développeurs et que ceux-ci sont en théorie à l'aise avec la langue anglaise, nous avons d'un commun accord de ne pas traduire les différents points de ce document de standards, nous avons seulement traduit les grands titres du document.

Source du document : http://symfony.com/doc/current/contributing/code/standards.html

1.2 Exemple de code respectant le standard de codage

```php
<?php

/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Acme;

/**
 * Coding standards demonstration.
 */
class FooBar
{
    const SOME_CONST = 42;

    /**
     * @var string
     */
    private $fooBar;

    /**
     * @param string $dummy Some argument description
     */
    public function __construct($dummy)
    {
        $this->fooBar = $this->transformText($dummy);
    }

    /**
     * @return string
     *
     * @deprecated
     */
    public function someDeprecatedMethod()
    {
        @trigger_error(sprintf('The %s() method is deprecated since version 2.8 and will be r
emoved in 3.0. Use Acme\Baz::someMethod() instead.', __METHOD__), E_USER_DEPRECATED);

        return Baz::someMethod();
    }

    /**
     * Transforms the input given as first argument.
     *
     * @param bool|string $dummy   Some argument description
     * @param array       $options An options collection to be used within the transformation
     *
     * @return string|null The transformed input
     *
     * @throws \RuntimeException When an invalid option is provided
     */
```

```php
    private function transformText($dummy, array $options = array())
    {
        $defaultOptions = array(
            'some_default' => 'values',
            'another_default' => 'more values',
        );

        foreach ($options as $option) {
            if (!in_array($option, $defaultOptions)) {
                throw new \RuntimeException(sprintf('Unrecognized option "%s"', $option));
            }
        }

        $mergedOptions = array_merge(
            $defaultOptions,
            $options
        );

        if (true === $dummy) {
            return null;
        }

        if ('string' === $dummy) {
            if ('values' === $mergedOptions['some_default']) {
                return substr($dummy, 0, 5);
            }

            return ucwords($dummy);
        }
    }

    /**
     * Performs some basic check for a given value.
     *
     * @param mixed $value     Some value to check against
     * @param bool  $theSwitch Some switch to control the method's flow
     *
     * @return bool|void The resultant check if $theSwitch isn't false, void otherwise
     */
    private function reverseBoolean($value = null, $theSwitch = false)
    {
        if (!$theSwitch) {
            return;
        }

        return !$value;
    }
}
```

1.3 Structure

- Add a single space after each comma delimiter;

- Add a single space around binary operators (==, &&, ...), with the exception of the concatenation (.) operator;

- Place unary operators (!, --, ...) adjacent to the affected variable;

- Always use identical comparison unless you need type juggling;

- Use Yoda conditions when checking a variable against an expression to avoid an accidental assignment inside the condition statement (this applies to ==, !=, ===, and !==);

- Add a comma after each array item in a multi-line array, even after the last one;

- Add a blank line before return statements, unless the return is alone inside a statement-group (like an if statement);

- Use return null; when a function explicitly returns null values and use return; when the function returns void values;

- Use braces to indicate control structure body regardless of the number of statements it contains;

- Define one class per file - this does not apply to private helper classes that are not intended to be instantiated from the outside and thus are not concerned by the PSR-0 and PSR-4 autoload standards;

- Declare the class inheritance and all the implemented interfaces on the same line as the class name;

- Declare class properties before methods;

- Declare public methods first, then protected ones and finally private ones. The exceptions to this rule are the class constructor and the setUp and tearDown methods of PHPUnit tests, which must always be the first methods to increase readability;

- Declare all the arguments on the same line as the method/function name, no matter how many arguments there are;

- Use parentheses when instantiating classes regardless of the number of arguments the constructor has;

- Exception and error message strings must be concatenated using sprintf;

- Calls to trigger_error with type E_USER_DEPRECATED must be switched to opt-in via @ operator. Read more at Deprecations;

- Do not use else, elseif, break after if and case conditions which return or throw something;

- Do not use spaces around [offset accessor and before] offset accessor.

1.4 Convention de nommage

- Use camelCase, not underscores, for variable, function and method names, arguments;

- Use underscores for option names and parameter names;

- Use namespaces for all classes;

- Prefix abstract classes with Abstract. Please note some early Symfony classes do not follow this convention and have not been renamed for backward compatibility reasons. However all new abstract classes must follow this naming convention;

- Suffix interfaces with Interface;

- Suffix traits with Trait;

- Suffix exceptions with Exception;

- Use alphanumeric characters and underscores for file names;

- For type-hinting in PHPDocs and casting, use bool (instead of boolean or Boolean), int (instead of integer), float (instead of double or real);

- Don't forget to look at the more verbose Conventions document for more subjective naming considerations.

1.5 Convention de nommage des services

- A service name contains groups, separated by dots;

- The DI alias of the bundle is the first group (e.g. fos_user);

- Use lowercase letters for service and parameter names;

- A group name uses the underscore notation.

1.6 Documentation

- Add PHPDoc[2] blocks for all classes, methods, and functions;

- Group annotations together so that annotations of the same type immediately follow each other, and annotations of a different type are separated by a single blank line;

- Omit the @return tag if the method does not return anything;

- The @package and @subpackage annotations are not used.

2. Glossaire

2.1 Symfony

Symfony est un framework MVC libre écrit en PHP (compatible avec PHP 7). Il fournit des fonctionnalités modulables et adaptables qui permettent de faciliter et d'accélérer le développement d'un site web.

2.2 PHPDoc

PHPDoc est une transposition de Javadoc au langage PHP. Il s'agit d'un standard formalisé pour commenter le code PHP. Il permet d'utiliser des outils tels que phpDocumentor ou Doxygen pour générer la documentation du code, notamment les méthodes publiques, les API, etc. Il permet aussi à certains IDE (Zend Studio (en), NetBeans, etc.) de connaître le type des variables et de lever d'autres ambiguïtés dues au typage faible, améliorant ainsi la complétion de code, le "typage objet" et le débogage.