How to contribute to Symfony

for Symfony 2.1

generated on November 25, 2013

Contents at a Glance

Chapter 1

Reporting a Bug

Whenever you find a bug in Symfony2, we kindly ask you to report it. It helps us make a better Symfony2.

If you think you've found a security issue, please use the special *procedure* instead.

Before submitting a bug:

- Double-check the official *documentation*[1] to see if you're not misusing the framework;
- Ask for assistance on the *users mailing-list*[2], the *forum*[3], or on the #symfony *IRC channel*[4] if you're not sure if your issue is really a bug.

If your problem definitely looks like a bug, report it using the official bug *tracker*[5] and follow some basic rules:

- Use the title field to clearly describe the issue;
- Describe the steps needed to reproduce the bug with short code examples (providing a unit test that illustrates the bug is best);
- Give as much detail as possible about your environment (OS, PHP version, Symfony version, enabled extensions, ...);
- *(optional)* Attach a *patch*.

1. `http://symfony.com/doc/current/`
2. `http://groups.google.com/group/symfony2`
3. `http://forum.symfony-project.org/`
4. `#contributing-code-irc:--irc.freenode.net-symfony`
5. `https://github.com/symfony/symfony/issues`

Chapter 2
Submitting a Patch

Patches are the best way to provide a bug fix or to propose enhancements to Symfony2.

Step 1: Setup your Environment

Install the Software Stack

Before working on Symfony2, setup a friendly environment with the following software:

- Git;
- PHP version 5.3.3 or above;
- PHPUnit 3.6.4 or above.

Configure Git

Set up your user information with your real name and a working email address:

Listing 2-1

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

If you are new to Git, you are highly recommended to read the excellent and free *ProGit*[1] book.

If your IDE creates configuration files inside the project's directory, you can use global `.gitignore` file (for all projects) or `.git/info/exclude` file (per project) to ignore them. See *Github's documentation*[2].

1. `http://git-scm.com/book`
2. `https://help.github.com/articles/ignoring-files`

Windows users: when installing Git, the installer will ask what to do with line endings, and suggests replacing all LF with CRLF. This is the wrong setting if you wish to contribute to Symfony! Selecting the as-is method is your best choice, as git will convert your line feeds to the ones in the repository. If you have already installed Git, you can check the value of this setting by typing:

Listing 2-2
```
$ git config core.autocrlf
```

This will return either "false", "input" or "true"; "true" and "false" being the wrong values. Change it to "input" by typing:

Listing 2-3
```
$ git config --global core.autocrlf input
```

Replace --global by --local if you want to set it only for the active repository

Get the Symfony Source Code

Get the Symfony2 source code:

- Create a *GitHub*[3] account and sign in;
- Fork the *Symfony2 repository*[4] (click on the "Fork" button);
- After the "forking action" has completed, clone your fork locally (this will create a *symfony* directory):

Listing 2-4
```
$ git clone git@github.com:USERNAME/symfony.git
```

- Add the upstream repository as a remote:

Listing 2-5
```
$ cd symfony
$ git remote add upstream git://github.com/symfony/symfony.git
```

Check that the current Tests pass

Now that Symfony2 is installed, check that all unit tests pass for your environment as explained in the dedicated *document*.

Step 2: Work on your Patch

The License

Before you start, you must know that all the patches you are going to submit must be released under the *MIT license*, unless explicitly specified in your commits.

3. `https://github.com/signup/free`
4. `https://github.com/symfony/symfony`

Choose the right Branch

Before working on a patch, you must determine on which branch you need to work. The branch should be based on the *master* branch if you want to add a new feature. But if you want to fix a bug, use the oldest but still maintained version of Symfony where the bug happens (like *2.1*).

All bug fixes merged into maintenance branches are also merged into more recent branches on a regular basis. For instance, if you submit a patch for the *2.1* branch, the patch will also be applied by the core team on the *master* branch.

Create a Topic Branch

Each time you want to work on a patch for a bug or on an enhancement, create a topic branch:

Listing 2-6
```
$ git checkout -b BRANCH_NAME master
```

Or, if you want to provide a bugfix for the 2.1 branch, first track the remote *2.1* branch locally:

Listing 2-7
```
$ git checkout -t origin/2.1
```

Then create a new branch off the 2.1 branch to work on the bugfix:

Listing 2-8
```
$ git checkout -b BRANCH_NAME 2.1
```

Use a descriptive name for your branch (*ticket_XXX* where *XXX* is the ticket number is a good convention for bug fixes).

The above checkout commands automatically switch the code to the newly created branch (check the branch you are working on with *git branch*).

Work on your Patch

Work on the code as much as you want and commit as much as you want; but keep in mind the following:

- Read about the Symfony *conventions* and follow the coding *standards* (use *git diff --check* to check for trailing spaces -- also read the tip below);
- Add unit tests to prove that the bug is fixed or that the new feature actually works;
- Try hard to not break backward compatibility (if you must do so, try to provide a compatibility layer to support the old way) -- patches that break backward compatibility have less chance to be merged;
- Do atomic and logically separate commits (use the power of *git rebase* to have a clean and logical history);
- Squash irrelevant commits that are just about fixing coding standards or fixing typos in your own code;
- Never fix coding standards in some existing code as it makes the code review more difficult;
- Write good commit messages (see the tip below).

You can check the coding standards of your patch by running the following *script*[5] (*source*[6]):

Listing 2-9

```
$ cd /path/to/symfony/src
$ php symfony-cs-fixer.phar fix . Symfony20Finder
```

A good commit message is composed of a summary (the first line), optionally followed by a blank line and a more detailed description. The summary should start with the Component you are working on in square brackets (`[DependencyInjection]`, `[FrameworkBundle]`, ...). Use a verb (`fixed ...`, `added ...`, ...) to start the summary and don't add a period at the end.

Prepare your Patch for Submission

When your patch is not about a bug fix (when you add a new feature or change an existing one for instance), it must also include the following:

- An explanation of the changes in the relevant CHANGELOG file(s) (the `[BC BREAK]` or the `[DEPRECATION]` prefix must be used when relevant);
- An explanation on how to upgrade an existing application in the relevant UPGRADE file(s) if the changes break backward compatibility or if you deprecate something that will ultimately break backward compatibility.

Step 3: Submit your Patch

Whenever you feel that your patch is ready for submission, follow the following steps.

Rebase your Patch

Before submitting your patch, update your branch (needed if it takes you a while to finish your changes):

Listing 2-10

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
$ git checkout BRANCH_NAME
$ git rebase master
```

Replace *master* with *2.1* if you are working on a bugfix

When doing the `rebase` command, you might have to fix merge conflicts. `git status` will show you the *unmerged* files. Resolve all the conflicts, then continue the rebase:

Listing 2-11

```
$ git add ... # add resolved files
$ git rebase --continue
```

Check that all tests still pass and push your branch remotely:

5. `http://cs.sensiolabs.org/get/php-cs-fixer.phar`
6. `https://github.com/fabpot/PHP-CS-Fixer`

Listing 2-12
```
$ git push origin BRANCH_NAME
```

Make a Pull Request

You can now make a pull request on the `symfony/symfony` Github repository.

Take care to point your pull request towards `symfony:2.1` if you want the core team to pull a bugfix based on the 2.1 branch.

To ease the core team work, always include the modified components in your pull request message, like in:

Listing 2-13
```
[Yaml] fixed something
[Form] [Validator] [FrameworkBundle] added something
```

The pull request description must include the following checklist at the top to ensure that contributions may be reviewed without needless feedback loops and that your contributions can be included into Symfony2 as quickly as possible:

Listing 2-14
```
| Q             | A
| ------------- | ---
| Bug fix?      | [yes|no]
| New feature?  | [yes|no]
| BC breaks?    | [yes|no]
| Deprecations? | [yes|no]
| Tests pass?   | [yes|no]
| Fixed tickets | [comma separated list of tickets fixed by the PR]
| License       | MIT
| Doc PR        | [The reference to the documentation PR if any]
```

An example submission could now look as follows:

Listing 2-15
```
| Q             | A
| ------------- | ---
| Bug fix?      | no
| New feature?  | no
| BC breaks?    | no
| Deprecations? | no
| Tests pass?   | yes
| Fixed tickets | #12, #43
| License       | MIT
| Doc PR        | symfony/symfony-docs#123
```

The whole table must be included (do **not** remove lines that you think are not relevant). For simple typos, minor changes in the PHPDocs, or changes in translation files, use the shorter version of the check-list:

Listing 2-16
```
| Q             | A
| ------------- | ---
| Fixed tickets | [comma separated list of tickets fixed by the PR]
| License       | MIT
```

Some answers to the questions trigger some more requirements:

- If you answer yes to "Bug fix?", check if the bug is already listed in the Symfony issues and reference it/them in "Fixed tickets";
- If you answer yes to "New feature?", you must submit a pull request to the documentation and reference it under the "Doc PR" section;
- If you answer yes to "BC breaks?", the patch must contain updates to the relevant CHANGELOG and UPGRADE files;
- If you answer yes to "Deprecations?", the patch must contain updates to the relevant CHANGELOG and UPGRADE files;
- If you answer no to "Tests pass", you must add an item to a todo-list with the actions that must be done to fix the tests;
- If the "license" is not MIT, just don't submit the pull request as it won't be accepted anyway.

If some of the previous requirements are not met, create a todo-list and add relevant items:

Listing 2-17

```
- [ ] fix the tests as they have not been updated yet
- [ ] submit changes to the documentation
- [ ] document the BC breaks
```

If the code is not finished yet because you don't have time to finish it or because you want early feedback on your work, add an item to todo-list:

Listing 2-18

```
- [ ] finish the code
- [ ] gather feedback for my changes
```

As long as you have items in the todo-list, please prefix the pull request title with "[WIP]".

In the pull request description, give as much details as possible about your changes (don't hesitate to give code examples to illustrate your points). If your pull request is about adding a new feature or modifying an existing one, explain the rationale for the changes. The pull request description helps the code review and it serves as a reference when the code is merged (the pull request description and all its associated comments are part of the merge commit message).

In addition to this "code" pull request, you must also send a pull request to the *documentation repository*[7] to update the documentation when appropriate.

Rework your Patch

Based on the feedback on the pull request, you might need to rework your patch. Before re-submitting the patch, rebase with `upstream/master` or `upstream/2.1`, don't merge; and force the push to the origin:

Listing 2-19

```
$ git rebase -f upstream/master
$ git push -f origin BRANCH_NAME
```

when doing a `push --force`, always specify the branch name explicitly to avoid messing other branches in the repo (`--force` tells git that you really want to mess with things so do it carefully).

Often, moderators will ask you to "squash" your commits. This means you will convert many commits to one commit. To do this, use the rebase command:

Listing 2-20

7. https://github.com/symfony/symfony-docs

```
$ git rebase -i HEAD~3
$ git push -f origin BRANCH_NAME
```

The number 3 here must equal the amount of commits in your branch. After you type this command, an editor will popup showing a list of commits:

Listing 2-21

```
pick 1a31be6 first commit
pick 7fc64b4 second commit
pick 7d33018 third commit
```

To squash all commits into the first one, remove the word "pick" before the second and the last commits, and replace it by the word "squash" or just "s". When you save, git will start rebasing, and if successful, will ask you to edit the commit message, which by default is a listing of the commit messages of all the commits. When you finish, execute the push command.

Chapter 3
Security Issues

This document explains how Symfony security issues are handled by the Symfony core team (Symfony being the code hosted on the main `symfony/symfony` Git repository).

Reporting a Security Issue

If you think that you have found a security issue in Symfony, don't use the mailing-list or the bug tracker and don't publish it publicly. Instead, all security issues must be sent to **security [at] symfony.com**. Emails sent to this address are forwarded to the Symfony core-team private mailing-list.

Resolving Process

For each report, we first try to confirm the vulnerability. When it is confirmed, the core-team works on a solution following these steps:

1. Send an acknowledgement to the reporter;
2. Work on a patch;
3. Get a CVE identifier from mitre.org;
4. Write a security announcement for the official Symfony *blog*[1] about the vulnerability. This post should contain the following information:

    - a title that always include the "Security release" string;
    - a description of the vulnerability;
    - the affected versions;
    - the possible exploits;
    - how to patch/upgrade/workaround affected applications;
    - the CVE identifier;
    - credits.

5. Send the patch and the announcement to the reporter for review;
6. Apply the patch to all maintained versions of Symfony;
7. Package new versions for all affected versions;

1. `http://symfony.com/blog/`

8. Publish the post on the official Symfony *blog*[2] (it must also be added to the "*Security Advisories*[3]" category);
9. Update the security advisory list (see below).

Releases that include security issues should not be done on Saturday or Sunday, except if the vulnerability has been publicly posted.

While we are working on a patch, please do not reveal the issue publicly.

The resolution takes anywhere between a couple of days to a month depending on its complexity and the coordination with the downstream projects (see next paragraph).

Collaborating with Downstream Open-Source Projects

As Symfony is used by many large Open-Source projects, we standardized the way the Symfony security team collaborates on security issues with downstream projects. The process works as follows:

1. After the Symfony security team has acknowledged a security issue, it immediately sends an email to the downstream project security teams to inform them of the issue;

2. The Symfony security team creates a private Git repository to ease the collaboration on the issue and access to this repository is given to the Symfony security team, to the Symfony contributors that are impacted by the issue, and to one representative of each downstream projects;

3. All people with access to the private repository work on a solution to solve the issue via pull requests, code reviews, and comments;

4. Once the fix is found, all involved projects collaborate to find the best date for a joint release (there is no guarantee that all releases will be at the same time but we will try hard to make them at about the same time). When the issue is not known to be exploited in the wild, a period of two weeks seems like a reasonable amount of time.

The list of downstream projects participating in this process is kept as small as possible in order to better manage the flow of confidential information prior to disclosure. As such, projects are included at the sole discretion of the Symfony security team.

As of today, the following projects have validated this process and are part of the downstream projects included in this process:

- Drupal (releases typically happen on Wednesdays)
- eZPublish

Security Advisories

This section indexes security vulnerabilities that were fixed in Symfony releases, starting from Symfony 1.0.0:

2. `http://symfony.com/blog/`
3. `http://symfony.com/blog/category/security-advisories`

- October 10, 2013: *Security releases: Symfony 2.0.25, 2.1.13, 2.2.9, and 2.3.6 released*[4] (*CVE-2013-5958*[5])
- August 7, 2013: *Security releases: Symfony 2.0.24, 2.1.12, 2.2.5, and 2.3.3 released*[6] (*CVE-2013-4751*[7] and *CVE-2013-4752*[8])
- January 17, 2013: *Security release: Symfony 2.0.22 and 2.1.7 released*[9] (*CVE-2013-1348*[10] and *CVE-2013-1397*[11])
- December 20, 2012: *Security release: Symfony 2.0.20 and 2.1.5*[12] (*CVE-2012-6431*[13] and *CVE-2012-6432*[14])
- November 29, 2012: *Security release: Symfony 2.0.19 and 2.1.4*[15]
- November 25, 2012: *Security release: symfony 1.4.20 released*[16] (*CVE-2012-5574*[17])
- August 28, 2012: *Security Release: Symfony 2.0.17 released*[18]
- May 30, 2012: *Security Release: symfony 1.4.18 released*[19] (*CVE-2012-2667*[20])
- February 24, 2012: *Security Release: Symfony 2.0.11 released*[21]
- November 16, 2011: *Security Release: Symfony 2.0.6*[22]
- March 21, 2011: *symfony 1.3.10 and 1.4.10: security releases*[23]
- June 29, 2010: *Security Release: symfony 1.3.6 and 1.4.6*[24]
- May 31, 2010: *symfony 1.3.5 and 1.4.5*[25]
- February 25, 2010: *Security Release: 1.2.12, 1.3.3 and 1.4.3*[26]
- February 13, 2010: *symfony 1.3.2 and 1.4.2*[27]
- April 27, 2009: *symfony 1.2.6: Security fix*[28]
- October 03, 2008: *symfony 1.1.4 released: Security fix*[29]
- May 14, 2008: *symfony 1.0.16 is out*[30]
- April 01, 2008: *symfony 1.0.13 is out*[31]
- March 21, 2008: *symfony 1.0.12 is (finally) out !*[32]
- June 25, 2007: *symfony 1.0.5 released (security fix)*[33]

4. `http://symfony.com/blog/security-releases-cve-2013-5958-symfony-2-0-25-2-1-13-2-2-9-and-2-3-6-released`
5. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5958`
6. `http://symfony.com/blog/security-releases-symfony-2-0-24-2-1-12-2-2-5-and-2-3-3-released`
7. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4751`
8. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4752`
9. `http://symfony.com/blog/security-release-symfony-2-0-22-and-2-1-7-released`
10. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1348`
11. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1397`
12. `http://symfony.com/blog/security-release-symfony-2-0-20-and-2-1-5-released`
13. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6431`
14. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6432`
15. `http://symfony.com/blog/security-release-symfony-2-0-19-and-2-1-4`
16. `http://symfony.com/blog/security-release-symfony-1-4-20-released`
17. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5574`
18. `http://symfony.com/blog/security-release-symfony-2-0-17-released`
19. `http://symfony.com/blog/security-release-symfony-1-4-18-released`
20. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2667`
21. `http://symfony.com/blog/security-release-symfony-2-0-11-released`
22. `http://symfony.com/blog/security-release-symfony-2-0-6`
23. `http://symfony.com/blog/symfony-1-3-10-and-1-4-10-security-releases`
24. `http://symfony.com/blog/security-release-symfony-1-3-6-and-1-4-6`
25. `http://symfony.com/blog/symfony-1-3-5-and-1-4-5`
26. `http://symfony.com/blog/security-release-1-2-12-1-3-3-and-1-4-3`
27. `http://symfony.com/blog/symfony-1-3-2-and-1-4-2`
28. `http://symfony.com/blog/symfony-1-2-6-security-fix`
29. `http://symfony.com/blog/symfony-1-1-4-released-security-fix`
30. `http://symfony.com/blog/symfony-1-0-16-is-out`
31. `http://symfony.com/blog/symfony-1-0-13-is-out`
32. `http://symfony.com/blog/symfony-1-0-12-is-finally-out`
33. `http://symfony.com/blog/symfony-1-0-5-released-security-fix`

Chapter 4
Running Symfony2 Tests

Before submitting a *patch* for inclusion, you need to run the Symfony2 test suite to check that you have not broken anything.

PHPUnit

To run the Symfony2 test suite, *install*[1] PHPUnit 3.6.4 or later first:

Listing 4-1
```
$ pear config-set auto_discover 1
$ pear install pear.phpunit.de/PHPUnit
```

Dependencies (optional)

To run the entire test suite, including tests that depend on external dependencies, Symfony2 needs to be able to autoload them. By default, they are autoloaded from *vendor/* under the main root directory (see *autoload.php.dist*).

The test suite needs the following third-party libraries:

- Doctrine
- Swiftmailer
- Twig
- Monolog

To install them all, use *Composer*[2]:

Step 1: Get *Composer*[3]

Listing 4-2
```
curl -s http://getcomposer.org/installer | php
```

1. http://www.phpunit.de/manual/current/en/installation.html
2. http://getcomposer.org/
3. http://getcomposer.org/

Make sure you download **composer.phar** in the same folder where the **composer.json** file is located.

Step 2: Install vendors

Listing 4-3
```
$ php composer.phar --dev install
```

Note that the script takes some time to finish.

If you don't have **curl** installed, you can also just download the **installer** file manually at *http://getcomposer.org/installer*[4]. Place this file into your project and then run:

Listing 4-4
```
$ php installer
$ php composer.phar --dev install
```

After installation, you can update the vendors to their latest version with the follow command:

Listing 4-5
```
$ php composer.phar --dev update
```

Running

First, update the vendors (see above).

Then, run the test suite from the Symfony2 root directory with the following command:

Listing 4-6
```
$ phpunit
```

The output should display *OK*. If not, you need to figure out what's going on and if the tests are broken because of your modifications.

If you want to test a single component type its path after the *phpunit* command, e.g.:

Listing 4-7
```
$ phpunit src/Symfony/Component/Finder/
```

Run the test suite before applying your modifications to check that they run fine on your configuration.

Code Coverage

If you add a new feature, you also need to check the code coverage by using the *coverage-html* option:

Listing 4-8

4. `http://getcomposer.org/installer`

```
$ phpunit --coverage-html=cov/
```

Check the code coverage by opening the generated *cov/index.html* page in a browser.

The code coverage only works if you have XDebug enabled and all dependencies installed.

Chapter 5
Coding Standards

When contributing code to Symfony2, you must follow its coding standards. To make a long story short, here is the golden rule: **Imitate the existing Symfony2 code**. Most open-source Bundles and libraries used by Symfony2 also follow the same guidelines, and you should too.

Remember that the main advantage of standards is that every piece of code looks and feels familiar, it's not about this or that being more readable.

Symfony follows the standards defined in the *PSR-0*[1], *PSR-1*[2] and *PSR-2*[3] documents.

Since a picture - or some code - is worth a thousand words, here's a short example containing most features described below:

Listing 5-1

```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;

    private $fooBar;
```

1. `https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md`
2. `https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md`
3. `https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md`

```php

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

    /**
     * @param string $dummy Some argument description
     * @param array  $options
     *
     * @return string|null Transformed input
     */
    private function transformText($dummy, array $options = array())
    {
        $mergedOptions = array_merge(
            $options,
            array(
                'some_default' => 'values',
                'another_default' => 'more values',
            )
        );

        if (true === $dummy) {
            return;
        }
        if ('string' === $dummy) {
            if ('values' === $mergedOptions['some_default']) {
                $dummy = substr($dummy, 0, 5);
            } else {
                $dummy = ucwords($dummy);
            }
        } else {
            throw new \RuntimeException(sprintf('Unrecognized dummy option "%s"', $dummy));
        }

        return $dummy;
    }
}
```

Structure

- Add a single space after each comma delimiter;
- Add a single space around operators (`==`, `&&`, ...);
- 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 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*[4] standard;
- Declare class properties before methods;

4. https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

- Declare public methods first, then protected ones and finally private ones;
- Use parentheses when instantiating classes regardless of the number of arguments the constructor has;
- Exception message strings should be concatenated using *`sprintf`*[5].

Naming Conventions

- 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 Symfony2 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;
- Don't forget to look at the more verbose *Conventions* document for more subjective naming considerations.

Documentation

- Add PHPDoc blocks for all classes, methods, and functions;
- Omit the `@return` tag if the method does not return anything;
- The `@package` and `@subpackage` annotations are not used.

License

- Symfony is released under the MIT license, and the license block has to be present at the top of every PHP file, before the namespace.

5. `http://php.net/manual/en/function.sprintf.php`

Chapter 6

Conventions

The *Coding Standards* document describes the coding standards for the Symfony2 projects and the internal and third-party bundles. This document describes coding standards and conventions used in the core framework to make it more consistent and predictable. You are encouraged to follow them in your own code, but you don't need to.

Method Names

When an object has a "main" many relation with related "things" (objects, parameters, ...), the method names are normalized:

- `get()`
- `set()`
- `has()`
- `all()`
- `replace()`
- `remove()`
- `clear()`
- `isEmpty()`
- `add()`
- `register()`
- `count()`
- `keys()`

The usage of these methods are only allowed when it is clear that there is a main relation:

- a `CookieJar` has many `Cookie` objects;
- a Service `Container` has many services and many parameters (as services is the main relation, the naming convention is used for this relation);
- a Console `Input` has many arguments and many options. There is no "main" relation, and so the naming convention does not apply.

For many relations where the convention does not apply, the following methods must be used instead (where `XXX` is the name of the related thing):

Main Relation	Other Relations
`get()`	`getXXX()`
`set()`	`setXXX()`
n/a	`replaceXXX()`
`has()`	`hasXXX()`
`all()`	`getXXXs()`
`replace()`	`setXXXs()`
`remove()`	`removeXXX()`
`clear()`	`clearXXX()`
`isEmpty()`	`isEmptyXXX()`
`add()`	`addXXX()`
`register()`	`registerXXX()`
`count()`	`countXXX()`
`keys()`	n/a

While "setXXX" and "replaceXXX" are very similar, there is one notable difference: "setXXX" may replace, or add new elements to the relation. "replaceXXX", on the other hand, cannot add new elements. If an unrecognized key as passed to "replaceXXX" it must throw an exception.

Deprecations

From time to time, some classes and/or methods are deprecated in the framework; that happens when a feature implementation cannot be changed because of backward compatibility issues, but we still want to propose a "better" alternative. In that case, the old implementation can simply be **deprecated**.

A feature is marked as deprecated by adding a `@deprecated` phpdoc to relevant classes, methods, properties, ...:

Listing 6-1

```
/**
 * @deprecated Deprecated since version 2.X, to be removed in 2.Y. Use XXX instead.
 */
```

The deprecation message should indicate the version when the class/method was deprecated, the version when it will be removed, and whenever possible, how the feature was replaced.

A PHP `E_USER_DEPRECATED` error must also be triggered to help people with the migration starting one or two minor versions before the version where the feature will be removed (depending on the criticality of the removal):

Listing 6-2

```
trigger_error(
    'XXX() is deprecated since version 2.X and will be removed in 2.Y. Use XXX instead.',
    E_USER_DEPRECATED
);
```

Chapter 7
Git

This document explains some conventions and specificities in the way we manage the Symfony code with Git.

Pull Requests

Whenever a pull request is merged, all the information contained in the pull request (including comments) is saved in the repository.

You can easily spot pull request merges as the commit message always follows this pattern:

Listing 7-1

```
merged branch USER_NAME/BRANCH_NAME (PR #1111)
```

The PR reference allows you to have a look at the original pull request on Github: *https://github.com/symfony/symfony/pull/1111*[1]. But all the information you can get on Github is also available from the repository itself.

The merge commit message contains the original message from the author of the changes. Often, this can help understand what the changes were about and the reasoning behind the changes.

Moreover, the full discussion that might have occurred back then is also stored as a Git note (before March 22 2013, the discussion was part of the main merge commit message). To get access to these notes, add this line to your `.git/config` file:

Listing 7-2

```
fetch = +refs/notes/*:refs/notes/*
```

After a fetch, getting the Github discussion for a commit is then a matter of adding `--show-notes=github-comments` to the `git show` command:

Listing 7-3

```
$ git show HEAD --show-notes=github-comments
```

1. `https://github.com/symfony/symfony/pull/1111`

Chapter 8
Symfony2 License

Symfony2 is released under the MIT license.

According to *Wikipedia*[1]:

> "It is a permissive license, meaning that it permits reuse within proprietary software on the condition that the license is distributed with that software. The license is also GPL-compatible, meaning that the GPL permits combination and redistribution with software that uses the MIT License."

The License

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

Chapter 9
Contributing to the Documentation

Documentation is as important as code. It follows the exact same principles: DRY, tests, ease of maintenance, extensibility, optimization, and refactoring just to name a few. And of course, documentation has bugs, typos, hard to read tutorials, and more.

Contributing

Before contributing, you need to become familiar with the *markup language* used by the documentation.

The Symfony2 documentation is hosted on GitHub:

Listing 9-1
```
https://github.com/symfony/symfony-docs
```

If you want to submit a patch, *fork*[1] the official repository on GitHub and then clone your fork:

Listing 9-2
```
$ git clone git://github.com/YOURUSERNAME/symfony-docs.git
```

Consistent with Symfony's source code, the documentation repository is split into multiple branches: `2.0`, `2.1`, `2.2` corresponding to the different versions of Symfony itself. The `master` branch holds the documentation for the development branch of the code.

Unless you're documenting a feature that was introduced *after* Symfony 2.1 (e.g. in Symfony 2.2), your changes should always be based on the 2.1 branch. To do this checkout the 2.1 branch before the next step:

Listing 9-3
```
$ git checkout 2.1
```

Your base branch (e.g. 2.1) will become the "Applies to" in the *Pull Request Format* that you'll use later.

1. `https://help.github.com/articles/fork-a-repo`

Next, create a dedicated branch for your changes (for organization):

Listing 9-4
```
$ git checkout -b improving_foo_and_bar
```

You can now make your changes directly to this branch and commit them. When you're done, push this branch to *your* GitHub fork and initiate a pull request.

Creating a Pull Request

Following the example, the pull request will default to be between your `improving_foo_and_bar` branch and the `symfony-docs` `master` branch.

../../_images/docs-pull-request.png

If you have made your changes based on the 2.1 branch then you need to change the base branch to be 2.1 on the preview page:

../../_images/docs-pull-request-change-base.png

All changes made to a branch (e.g. 2.1) will be merged up to each "newer" branch (e.g. 2.2, master, etc) for the next release on a weekly basis.

GitHub covers the topic of *pull requests*[2] in detail.

You can also prefix the title of your pull request in a few cases:

- `[WIP]` (Work in Progress) is used when you are not yet finished with your pull request, but you would like it to be reviewed. The pull request won't be merged until you say it is ready.
- `[WCM]` (Waiting Code Merge) is used when you're documenting a new feature or change that hasn't been accepted yet into the core code. The pull request will not be merged until it is merged in the core code (or closed if the change is rejected).

Pull Request Format

Unless you're fixing some minor typos, the pull request description **must** include the following checklist to ensure that contributions may be reviewed without needless feedback loops and that your contributions can be included into the documentation as quickly as possible:

Listing 9-5
```
| Q             | A
| ------------- | ---
| Doc fix?      | [yes|no]
| New docs?     | [yes|no] (PR # on symfony/symfony if applicable)
| Applies to    | [Symfony version numbers this applies to]
| Fixed tickets | [comma separated list of tickets fixed by the PR]
```

An example submission could now look as follows:

2. https://help.github.com/articles/using-pull-requests

Listing 9-6

```
| Q             | A
| ------------- | ---
| Doc fix?      | yes
| New docs?     | yes (symfony/symfony#2500)
| Applies to    | all (or 2.1+)
| Fixed tickets | #1075
```

Please be patient. It can take from 15 minutes to several days for your changes to appear on the symfony.com website after the documentation team merges your pull request. You can check if your changes have introduced some markup issues by going to the *Documentation Build Errors*[3] page (it is updated each French night at 3AM when the server rebuilds the documentation).

Documenting new Features or Behavior Changes

If you're documenting a brand new feature or a change that's been made in Symfony2, you should precede your description of the change with a `.. versionadded:: 2.X` tag and a short description:

Listing 9-7

```
.. versionadded:: 2.2
    The ``askHiddenResponse`` method was added in Symfony 2.2.

You can also ask a question and hide the response. This is particularly...
```

If you're documenting a behavior change, it may be helpful to *briefly* describe how the behavior has changed.

Listing 9-8

```
.. versionadded:: 2.2
    The ``include()`` function is a new Twig feature that's available in
    Symfony 2.2. Prior, the ``{% include %}`` tag was used.
```

Whenever a new minor version of Symfony2 is released (e.g. 2.3, 2.4, etc), a new branch of the documentation is created from the `master` branch. At this point, all the `versionadded` tags for Symfony2 versions that have reached end-of-life will be removed. For example, if Symfony 2.5 were released today, and 2.2 had recently reached its end-of-life, the 2.2 `versionadded` tags would be removed from the new 2.5 branch.

Standards

All documentation in the Symfony Documentation should follow *the documentation standards*.

Reporting an Issue

The most easy contribution you can make is reporting issues: a typo, a grammar mistake, a bug in a code example, a missing explanation, and so on.

Steps:

- Submit a bug in the bug tracker;
- *(optional)* Submit a patch.

3. `http://symfony.com/doc/build_errors`

Translating

Read the dedicated *document*.

Chapter 10
Documentation Format

The Symfony2 documentation uses *reStructuredText*[1] as its markup language and *Sphinx*[2] for building the output (HTML, PDF, ...).

reStructuredText

reStructuredText "is an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax and parser system".

You can learn more about its syntax by reading existing Symfony2 *documents*[3] or by reading the *reStructuredText Primer*[4] on the Sphinx website.

If you are familiar with Markdown, be careful as things are sometimes very similar but different:

- Lists starts at the beginning of a line (no indentation is allowed);
- Inline code blocks use double-ticks (` ``like this`` `).

Sphinx

Sphinx is a build system that adds some nice tools to create documentation from reStructuredText documents. As such, it adds new directives and interpreted text roles to standard reST *markup*[5].

Syntax Highlighting

All code examples uses PHP as the default highlighted language. You can change it with the `code-block` directive:

Listing 10-1

1. `http://docutils.sourceforge.net/rst.html`
2. `http://sphinx-doc.org/`
3. `https://github.com/symfony/symfony-docs`
4. `http://sphinx-doc.org/rest.html`
5. `http://sphinx-doc.org/markup/`

```
.. code-block:: yaml

    { foo: bar, bar: { foo: bar, bar: baz } }
```

If your PHP code begins with `<?php`, then you need to use `html+php` as the highlighted pseudo-language:

Listing 10-2
```
.. code-block:: html+php

    <?php echo $this->foobar(); ?>
```

A list of supported languages is available on the *Pygments website*[6].

Configuration Blocks

Whenever you show a configuration, you must use the `configuration-block` directive to show the configuration in all supported configuration formats (`PHP`, `YAML`, and `XML`)

Listing 10-3
```
.. configuration-block::

    .. code-block:: yaml

        # Configuration in YAML

    .. code-block:: xml

        <!-- Configuration in XML //-->

    .. code-block:: php

        // Configuration in PHP
```

The previous reST snippet renders as follow:

Listing 10-4
```
# Configuration in YAML
```

The current list of supported formats are the following:

Markup format	**Displayed**
html	HTML
xml	XML
php	PHP
yaml	YAML
jinja	Twig
html+jinja	Twig
html+php	PHP

6. `http://pygments.org/languages/`

Markup format	Displayed
ini	INI
php-annotations	Annotations

Adding Links

To add links to other pages in the documents use the following syntax:

Listing 10-5
```
:doc:`/path/to/page`
```

Using the path and filename of the page without the extension, for example:

Listing 10-6
```
:doc:`/book/controller`

:doc:`/components/event_dispatcher/introduction`

:doc:`/cookbook/configuration/environments`
```

The link text will be the main heading of the document linked to. You can also specify alternative text for the link:

Listing 10-7
```
:doc:`Spooling Email</cookbook/email/spool>`
```

You can also add links to the API documentation:

Listing 10-8
```
:namespace:`Symfony\\Component\\BrowserKit`

:class:`Symfony\\Component\\Routing\\Matcher\\ApacheUrlMatcher`

:method:`Symfony\\Component\\HttpKernel\\Bundle\\Bundle::build`
```

and to the PHP documentation:

Listing 10-9
```
:phpclass:`SimpleXMLElement`

:phpmethod:`DateTime::createFromFormat`

:phpfunction:`iterator_to_array`
```

Testing Documentation

To test documentation before a commit:

- Install *Sphinx*[7];
- Run the *Sphinx quick setup*[8];
- Install the Sphinx extensions (see below);
- Run `make html` and view the generated HTML in the `build` directory.

7. `http://sphinx-doc.org/`
8. `http://sphinx-doc.org/tutorial.html#setting-up-the-documentation-sources`

Installing the Sphinx extensions

- Download the extension from the *source*[9] repository
- Copy the `sensio` directory to the `_exts` folder under your source folder (where `conf.py` is located)
- Add the following to the `conf.py` file:

Listing 10-10

```
# ...
sys.path.append(os.path.abspath('_exts'))

# adding PhpLexer
from sphinx.highlighting import lexers
from pygments.lexers.web import PhpLexer

# ...
# add the extensions to the list of extensions
extensions = [..., 'sensio.sphinx.refinclude', 'sensio.sphinx.configurationblock',
'sensio.sphinx.phpcode']

# enable highlighting for PHP code not between ``<?php ... ?>`` by default
lexers['php'] = PhpLexer(startinline=True)
lexers['php-annotations'] = PhpLexer(startinline=True)

# use PHP as the primary domain
primary_domain = 'php'

# set url for API links
api_url = 'http://api.symfony.com/master/%s'
```

9. `https://github.com/fabpot/sphinx-php`

Chapter 11
Documentation Standards

In order to help the reader as much as possible and to create code examples that look and feel familiar, you should follow these standards.

Sphinx

- The following characters are choosen for different heading levels: level 1 is `=`, level 2 `-`, level 3 `~`, level 4 `.` and level 5 `"`;
- Each line should break approximately after the first word that crosses the 72nd character (so most lines end up being 72-78 characters);
- The `::` shorthand is *preferred* over `.. code-block:: php` to begin a PHP code block (read *the Sphinx documentation*[1] to see when you should use the shorthand);
- Inline hyperlinks are **not** used. Seperate the link and their target definition, which you add on the bottom of the page;
- Inline markup should be closed on the same line as the open-string;
- You should use a form of *you* instead of *we*.

Example

Listing 11-1

```
Example
=======

When you are working on the docs, you should follow the
`Symfony Documentation`_ standards.

Level 2
-------

A PHP example would be::

    echo 'Hello World';
```

1. `http://sphinx-doc.org/rest.html#source-code`

```

Level 3
~~~~~~~

.. code-block:: php

    echo 'You cannot use the :: shortcut here';

.. _`Symfony Documentation`: http://symfony.com/doc/current/contributing/documentation/
standards.html
```

Code Examples

- The code follows the *Symfony Coding Standards* as well as the *Twig Coding Standards*[2];
- To avoid horizontal scrolling on code blocks, we prefer to break a line correctly if it crosses the 85th character;
- When you fold one or more lines of code, place `...` in a comment at the point of the fold. These comments are: `// ...` (php), `# ...` (yaml/bash), `{# ... #}` (twig), `<!-- ... -->` (xml/html), `; ...` (ini), `...` (text);
- When you fold a part of a line, e.g. a variable value, put `...` (without comment) at the place of the fold;
- Description of the folded code: (optional) If you fold several lines: the description of the fold can be placed after the `...` If you fold only part of a line: the description can be placed before the line;
- If useful to the reader, a PHP code example should start with the namespace declaration;
- When referencing classes, be sure to show the `use` statements at the top of your code block. You don't need to show *all* `use` statements in every example, just show what is actually being used in the code block;
- If useful, a `codeblock` should begin with a comment containing the filename of the file in the code block. Don't place a blank line after this comment, unless the next line is also a comment;
- You should put a `$` in front of every bash line.

Formats

Configuration examples should show all supported formats using *configuration blocks*. The supported formats (and their orders) are:

- **Configuration** (including services and routing): Yaml, Xml, Php
- **Validation**: Yaml, Annotations, Xml, Php
- **Doctrine Mapping**: Annotations, Yaml, Xml, Php

Example

Listing 11-2

```php
// src/Foo/Bar.php
namespace Foo;

use Acme\Demo\Cat;
// ...

```

2. `http://twig.sensiolabs.org/doc/coding_standards.html`

```php
class Bar
{
    // ...

    public function foo($bar)
    {
        // set foo with a value of bar
        $foo = ...;

        $cat = new Cat($foo);

        // ... check if $bar has the correct value

        return $cat->baz($bar, ...);
    }
}
```

In Yaml you should put a space after `{` and before `}` (e.g. `{ _controller: ... }`), but this should not be done in Twig (e.g. `{'hello' : 'value'}`).

Chapter 12

Translations

The Symfony2 documentation is written in English and many people are involved in the translation process.

Contributing

First, become familiar with the *markup language* used by the documentation.

Then, subscribe to the *Symfony docs mailing-list*[1], as collaboration happens there.

Finally, find the *master* repository for the language you want to contribute for. Here is the list of the official *master* repositories:

- *English*: *https://github.com/symfony/symfony-docs*[2]
- *French*: *https://github.com/symfony-fr/symfony-docs-fr*[3]
- *Italian*: *https://github.com/garak/symfony-docs-it*[4]
- *Japanese*: *https://github.com/symfony-japan/symfony-docs-ja*[5]
- *Polish*: *https://github.com/symfony-docs-pl/symfony-docs-pl*[6]
- *Portuguese (Brazilian)*: *https://github.com/andreia/symfony-docs-pt-BR*[7]
- *Romanian*: *https://github.com/sebio/symfony-docs-ro*[8]
- *Russian*: *https://github.com/avalanche123/symfony-docs-ru*[9]
- *Spanish*: *https://github.com/gitnacho/symfony-docs-es*[10]
- *Turkish*: *https://github.com/symfony-tr/symfony-docs-tr*[11]

1. `http://groups.google.com/group/symfony-docs`
2. `https://github.com/symfony/symfony-docs`
3. `https://github.com/symfony-fr/symfony-docs-fr`
4. `https://github.com/garak/symfony-docs-it`
5. `https://github.com/symfony-japan/symfony-docs-ja`
6. `https://github.com/symfony-docs-pl/symfony-docs-pl`
7. `https://github.com/andreia/symfony-docs-pt-BR`
8. `https://github.com/sebio/symfony-docs-ro`
9. `https://github.com/avalanche123/symfony-docs-ru`
10. `https://github.com/gitnacho/symfony-docs-es`
11. `https://github.com/symfony-tr/symfony-docs-tr`

If you want to contribute translations for a new language, read the *dedicated section*.

Joining the Translation Team

If you want to help translating some documents for your language or fix some bugs, consider joining us; it's a very easy process:

- Introduce yourself on the *Symfony docs mailing-list*[12];
- *(optional)* Ask which documents you can work on;
- Fork the *master* repository for your language (click the "Fork" button on the GitHub page);
- Translate some documents;
- Ask for a pull request (click on the "Pull Request" from your page on GitHub);
- The team manager accepts your modifications and merges them into the master repository;
- The documentation website is updated every other night from the master repository.

Adding a new Language

This section gives some guidelines for starting the translation of the Symfony2 documentation for a new language.

As starting a translation is a lot of work, talk about your plan on the *Symfony docs mailing-list*[13] and try to find motivated people willing to help.

When the team is ready, nominate a team manager; he will be responsible for the *master* repository.

Create the repository and copy the *English* documents.

The team can now start the translation process.

When the team is confident that the repository is in a consistent and stable state (everything is translated, or non-translated documents have been removed from the toctrees -- files named `index.rst` and `map.rst.inc`), the team manager can ask that the repository is added to the list of official *master* repositories by sending an email to Fabien (fabien at symfony.com).

Maintenance

Translation does not end when everything is translated. The documentation is a moving target (new documents are added, bugs are fixed, paragraphs are reorganized, ...). The translation team need to closely follow the English repository and apply changes to the translated documents as soon as possible.

Non maintained languages are removed from the official list of repositories as obsolete documentation is dangerous.

12. `http://groups.google.com/group/symfony-docs`
13. `http://groups.google.com/group/symfony-docs`

Chapter 13

Symfony2 Documentation License

1. `http://creativecommons.org/licenses/by-sa/3.0/`
2. `http://creativecommons.org/licenses/by-sa/3.0/legalcode`

Chapter 14
The Release Process

This document explains the Symfony release process (Symfony being the code hosted on the main `symfony/symfony` *Git repository*[1]).

Symfony manages its releases through a *time-based model*; a new Symfony release comes out every *six months*: one in *May* and one in *November*.

This release process has been adopted as of Symfony 2.2, and all the "rules" explained in this document must be strictly followed as of Symfony 2.4.

Development

The six-months period is divided into two phases:

- *Development*: *Four months* to add new features and to enhance existing ones;
- *Stabilisation*: *Two months* to fix bugs, prepare the release, and wait for the whole Symfony ecosystem (third-party libraries, bundles, and projects using Symfony) to catch up.

During the development phase, any new feature can be reverted if it won't be finished in time or if it won't be stable enough to be included in the current final release.

Maintenance

Each Symfony version is maintained for a fixed period of time, depending on the type of the release. We have two maintenance periods:

- *Bug fixes and security fixes*: During this period, all issues can be fixed. The end of this period is referenced as being the *end of maintenance* of a release.
- *Security fixes only*: During this period, only security related issues can be fixed. The end of this period is referenced as being the *end of life* of a release.

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

Standard Releases

A standard release is maintained for an *eight month* period for bug fixes, and for a *fourteen month* period for security issue fixes.

Long Term Support Releases

Every two years, a new Long Term Support Release (aka LTS release) is published. Each LTS release is supported for a *three year* period for bug fixes, and for a *four year* period for security issue fixes.

Paid support after the three year support provided by the community can also be bought from *SensioLabs*[2].

Schedule

Below is the schedule for the first few versions that use this release model:

../../_images/release-process.jpg

- **Yellow** represents the Development phase
- **Blue** represents the Stabilisation phase
- **Green** represents the Maintenance period

This results in very predictable dates and maintenance periods:

Version	Release	End of Maintenance	End of Life
2.0	07/2011	03/2013 (20 months)	09/2013
2.1	09/2012	05/2013 (9 months)	11/2013
2.2	03/2013	11/2013 (8 months)	05/2014
2.3	05/2013	05/2016 (36 months)	05/2017
2.4	11/2013	07/2014 (8 months)	01/2015
2.5	05/2014	01/2015 (8 months)	07/2016
2.6	11/2014	07/2015 (8 months)	01/2016
2.7	05/2015	05/2018 (36 months)	05/2019
2.8	11/2015	07/2016 (8 months)	01/2017
...	...	...	...

If you want to learn more about the timeline of any given Symfony version, use the online *timeline calculator*[3]. You can also get all data as a JSON string via a URL like *http://symfony.com/roadmap.json?version=2.x*.

2. `http://sensiolabs.com/`
3. `http://symfony.com/roadmap`

Backward Compatibility

After the release of Symfony 2.3, backward compatibility will be kept at all cost. If it is not possible, the feature, the enhancement, or the bug fix will be scheduled for the next major version: Symfony 3.0.

The work on Symfony 3.0 will start whenever enough major features breaking backward compatibility are waiting on the todo-list.

Deprecations

When a feature implementation cannot be replaced with a better one without breaking backward compatibility, there is still the possibility to deprecate the old implementation and add a new preferred one along side. Read the *conventions* document to learn more about how deprecations are handled in Symfony.

Rationale

This release process was adopted to give more *predictability* and *transparency*. It was discussed based on the following goals:

- Shorten the release cycle (allow developers to benefit from the new features faster);
- Give more visibility to the developers using the framework and Open-Source projects using Symfony;
- Improve the experience of Symfony core contributors: everyone knows when a feature might be available in Symfony;
- Coordinate the Symfony timeline with popular PHP projects that work well with Symfony and with projects using Symfony;
- Give time to the Symfony ecosystem to catch up with the new versions (bundle authors, documentation writers, translators, ...).

The six month period was chosen as two releases fit in a year. It also allows for plenty of time to work on new features and it allows for non-ready features to be postponed to the next version without having to wait too long for the next cycle.

The dual maintenance mode was adopted to make every Symfony user happy. Fast movers, who want to work with the latest and the greatest, use the standard releases: a new version is published every six months, and there is a two months period to upgrade. Companies wanting more stability use the LTS releases: a new version is published every two years and there is a year to upgrade.

Chapter 15
IRC Meetings

The purpose of this meeting is to discuss topics in real time with many of the Symfony2 devs.

Anyone may propose topics on the *symfony-dev*[1] mailing-list until 24 hours before the meeting, ideally including well prepared relevant information via some URL. 24 hours before the meeting a link to a *doodle*[2] will be posted including a list of all proposed topics. Anyone can vote on the topics until the beginning of the meeting to define the order in the agenda. Each topic will be timeboxed to 15mins and the meeting lasts one hour, leaving enough time for at least 4 topics.

Note that it's not the expected goal of the meeting to find final solutions, but more to ensure that there is a common understanding of the issue at hand and move the discussion forward in ways which are hard to achieve with less real time communication tools.

Meetings will happen each Thursday at 17:00 CET (+01:00) on the #symfony-dev channel on the Freenode IRC server.

The IRC *logs*[3] will later be published on the trac wiki, which will include a short summary for each of the topics. Tickets will be created for any tasks or issues identified during the meeting and referenced in the summary.

Some simple guidelines and pointers for participation:

- It's possible to change votes until the beginning of the meeting by clicking on "Edit an entry";
- The doodle will be closed for voting at the beginning of the meeting;
- Agenda is defined by which topics got the most votes in the doodle, or whichever was proposed first in case of a tie;
- At the beginning of the meeting one person will identify him/herself as the moderator;
- The moderator is essentially responsible for ensuring the 15min timebox and ensuring that tasks are clearly identified;
- Usually the moderator will also handle writing the summary and creating trac tickets unless someone else steps up;
- Anyone can join and is explicitly invited to participate;
- Ideally one should familiarize oneself with the proposed topic before the meeting;

1. `http://groups.google.com/group/symfony-devs`
2. `http://doodle.com`
3. `http://trac.symfony-project.org/wiki/Symfony2IRCMeetingLogs`

- When starting on a new topic the proposer is invited to start things off with a few words;
- Anyone can then comment as they see fit;
- Depending on how many people participate one should potentially retrain oneself from pushing a specific argument too hard;
- Remember the IRC *logs*[4] will be published later on, so people have the chance to review comments later on once more;
- People are encouraged to raise their hand to take on tasks defined during the meeting.

Here is an *example*[5] doodle.

4. `http://trac.symfony-project.org/wiki/Symfony2IRCMeetingLogs`

5. `http://doodle.com/4cnzme7xys3ay53w`

Chapter 16

Other Resources

In order to follow what is happening in the community you might find helpful these additional resources:

- List of open *pull requests*[1]
- List of recent *commits*[2]
- List of open *bugs and enhancements*[3]
- List of open source *bundles*[4]

1. `https://github.com/symfony/symfony/pulls`
2. `https://github.com/symfony/symfony/commits/master`
3. `https://github.com/symfony/symfony/issues`
4. `http://knpbundles.com/`