

# Getting started with Symfony4

SF4C1

**SensioLabs**

# License

## Getting Started with Symfony 4

Copyright © 2018 SensioLabs – All Rights Reserved

Contents are provided for strictly personal use. No part of these contents may be reproduced, stored in a retrieval system, publicly shared or transmitted in any form or by any means.

## Disclaimer

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

# What is Symfony?

# The Symphony Project

# The Symfony Project



=

components  
+  
framework

# Symfony Components

Symfony is a **reusable** set of **standalone**, **decoupled**, and cohesive PHP 7.1 **components** that solve common web development problems.

# Symfony Full-Stack Framework

Symfony is also a **full-stack PHP framework** developed with the Symfony Components.

# List of Symfony Components

- Asset
- BrowserKit
- Cache
- Config
- Console
- CssSelector
- Debug
- DependencyInjection
- DomCrawler
- Dotenv
- EventDispatcher
- ExpressionLanguage
- Filesystem
- Finder
- Form
- HttpFoundation
- HttpKernel
- Intl
- Ldap
- Lock
- Messenger
- Mime
- OptionsResolver
- Process
- PropertyAccess
- PropertyInfo
- Routing
- Security
- Serializer
- Stopwatch
- Templating
- Translation
- Validator
- VarDumper
- VarExporter
- WebLink
- Workflow
- Yaml
- PHPUnit Bridge
- Polyfill APCu
- Polyfill PHP 5.4
- Polyfill PHP 5.5
- Polyfill PHP 5.6
- Polyfill PHP 7.0
- Polyfill PHP 7.1
- Polyfill PHP 7.2
- Polyfill PHP 7.3
- Polyfill Iconv
- Polyfill Intl
- Polyfill Mbstring
- Polyfill Util
- Polyfill Xml

# Symfony Source Code

# Symfony Source Code

- The official repository is hosted at GitHub  
<https://github.com/symfony/symfony>
- Big community, but clear vision  
Features proposed by thousands of developers but reviewed and accepted by a Core Team
- It's published under MIT License  
Permissive, business-friendly and GPL compatible

# Symfony Lifecycle

# Symfony development

- New versions are released on a **time-based schedule** (not on a feature-based schedule)
- **Semantic versioning** is followed strictly (your apps won't break during minor upgrades)
- This makes Symfony dependable and safe for companies.

More details about SemVer (semantic versioning): [semver.org](http://semver.org)

# Symfony releases

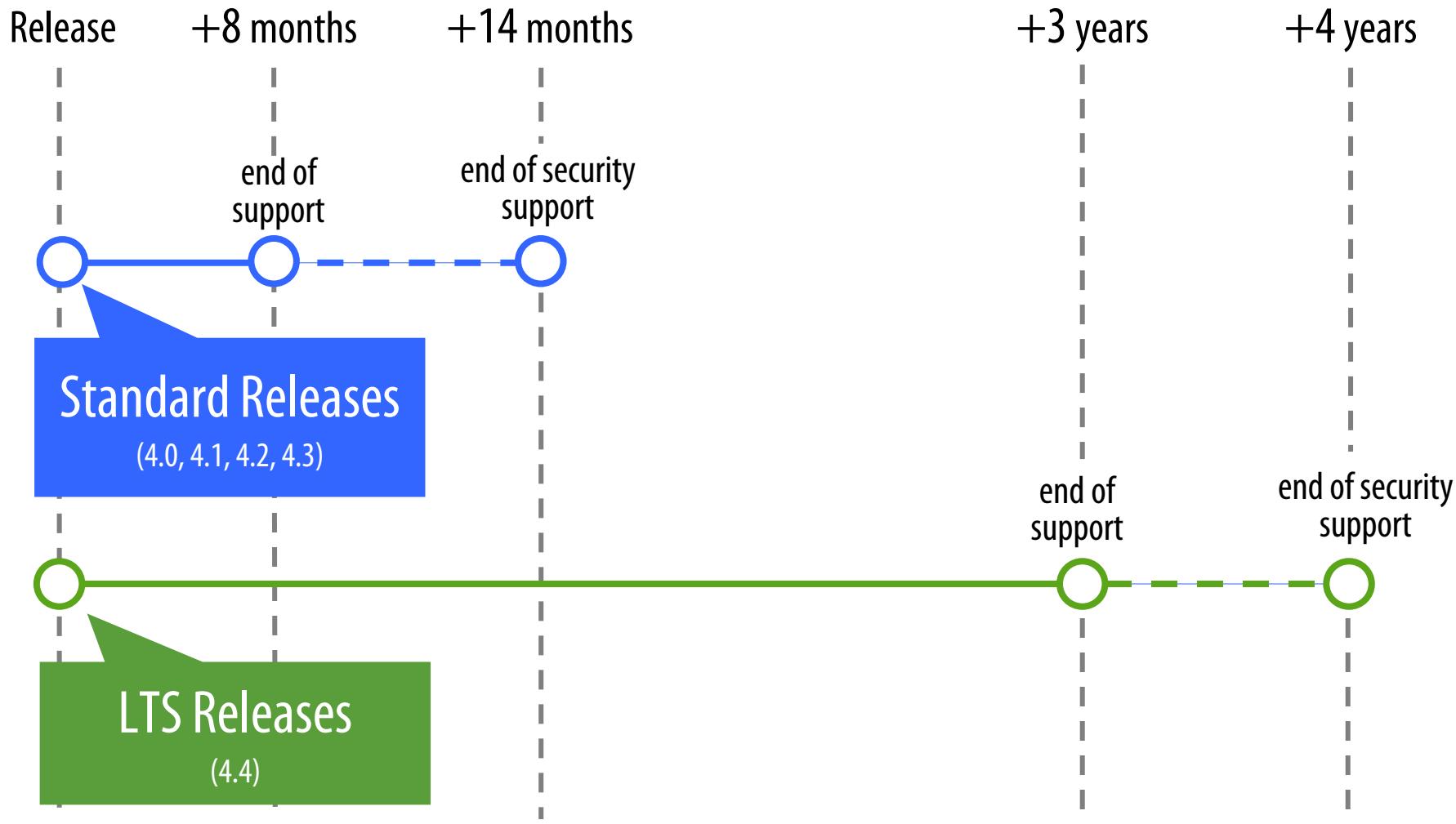
- **Patch versions** (X.Y.1, X.Y.2, X.Y.3, etc.)  
released monthly
- **Minor versions** (X.1.0, X.2.0, X.3.0, X.4.0)  
released twice a year (May and November)  
each major version releases 4 minor versions
- **Major versions** (3.0.0, 4.0.0, 5.0.0, etc.)  
released every two years

**TIP** Subscribe for free to receive email notifications when minor and major versions are released and/or deprecated: [symfony.com/roadmap](https://symfony.com/roadmap)

# Symfony support

- Standard versions
  - 8 months of bug support
  - 14 months for security support
- Long Term Support versions (LTS)
  - Last version of the branch: 3.4, 4.4, 5.4, etc.
  - 3 years of bug support
  - 4 years of security support

# Symfony Lifecycle



# Integration with developer tools

# IDEs and text editors

## Text editors



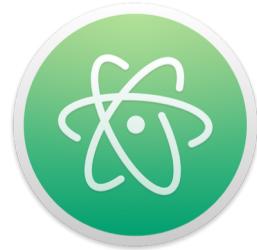
SublimeText



Vim



TextMate



Atom

## Full-featured IDEs



the most popular option for Symfony developers

# Symfony Resources

# Helpful Resources

- Official documentation
  - [symfony.com/doc](https://symfony.com/doc)
- Official support channels
  - [symfony.com/support](https://symfony.com/support)
- Report issues or ask for new features
  - [github.com/symfony/symfony](https://github.com/symfony/symfony)

# Resources to stay updated about Symfony

- Official Symfony Blog ([symfony.com/blog](https://symfony.com/blog))  
News, announcements and "New in Symfony" posts
- Community Events ([symfony.com/events](https://symfony.com/events))  
Meetups, conferences, hackathons, etc.
- Twitter
  - [@symfony](https://twitter.com/symfony)
  - [@symfony\\_live](https://twitter.com/symfony_live)
  - [@symfonydocs](https://twitter.com/symfonydocs)
  - [@symfonycon](https://twitter.com/symfonycon)



# Installing Symfony

# Symfony Installation

- Symfony is not installed once in your system and reused in all your projects.
- Instead, you **create** a small and empty Symfony application whenever you create a new project.

# Symfony Installation

- In the past, Symfony provided a small utility called **Symfony Installer**.
- In Symfony 4, this installer is **no longer recommended** and you must use **Composer** instead.

# Composer

# What is Composer?

Composer is the dependency manager used by all modern PHP applications.

Official website: [getcomposer.org](http://getcomposer.org)

## Best-practice

Composer should be installed  
globally in your system.

# Installing Composer

- **Manual installation:**

[getcomposer.org/download](https://getcomposer.org/download)

- **Programmatic installation:**

[getcomposer.org/doc/faqs/how-to-install-composer-programmatically.md](https://getcomposer.org/doc/faqs/how-to-install-composer-programmatically.md)

# Updating Composer

```
$ composer self-update
```

or

```
$ sudo composer self-update
```

# Composer configuration files

- **composer.json**

The dependencies + **approximate versions** that the project wants to be installed.

- **composer.lock**

The dependencies + **exact versions** that were installed after resolving all the dependencies.

# Composer Version Constraints

"symfony/console": "4.3.2"

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Composer Version Constraints

"symfony/console": "4.3.\*"

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Composer Version Constraints

"symfony/console": "**~4.3**"

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Composer Version Constraints

`"symfony/console": "~4.3.1"`

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Composer Version Constraints

`"symfony/console": "^4.3"`

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Composer Version Constraints

`"symfony/console": "^4.3.1"`

3.3.0	3.4.0	4.0.0	4.1.0	4.2.0	4.3.0	4.4.0	5.0.0	5.1.0
3.3.1	3.4.1	4.0.1	4.1.1	4.2.1	4.3.1	4.4.1	5.0.1	5.1.1
3.3.2	3.4.2	4.0.2	4.1.2	4.2.2	4.3.2	4.4.2	5.0.2	5.1.2
3.3.3	3.4.3	4.0.3	4.1.3	4.2.3	4.3.3	4.4.3	5.0.3	5.1.3
3.3.4	3.4.4	4.0.4	4.1.4	4.2.4	4.3.4	4.4.4	5.0.4	5.1.4

# Differences

## Tilde (~)

$\sim 1.2 \implies >=1.2.0 <2.0.0$

$\sim 1.2.3 \implies >=1.2.3 <1.3.0$

## Caret (^)

$\wedge 1.2 \implies >=1.2.0 <2.0.0$

$\wedge 1.2.3 \implies >=1.2.3 <2.0.0$

# Creating a new Symfony project



# Generic Symfony project

```
$ composer create-project \
symfony/skeleton \
my-project
```

It installs the minimum required dependencies to create a Symfony project. Ideal for creating CLI apps, microservices, APIs, backends, etc.

# Symfony web application

```
$ composer create-project \
symfony/website-skeleton \
my-project
```

It installs all the dependencies commonly used in web applications. The resulting project size is much bigger.

## Based on a specific branch

```
$ composer create-project \
symfony/skeleton:^4.0 \
my-project
```

```
$ composer create-project \
symfony/skeleton:^4.2 \
my-project
```

# Based on a specific version

```
$ composer create-project \
symfony/skeleton:^4.0.0 \
my-project
```

```
$ composer create-project \
symfony/skeleton:^4.2.4 \
my-project
```

# Check the installed Symfony version



# Display the installed Symfony version

```
$ cd my-project/  
$ php bin/console --version  
Symfony version 4.2.0
```



# Check if your system is ready for Symfony

```
$ cd my-project/  
$ composer require --dev \  
requirements-checker
```

...

Executing requirements-checker [OK]

```
$ vendor/bin/requirements-checker  
Symfony Requirements Checker
```

# Installing an existing Symfony project

# Install an existing Symfony project

```
$ cd projects/  
$ git clone .../my-project.git  
$ cd my-project/  
$ composer install
```

# Updating an existing Symfony project

# Update an existing Symfony project

```
$ cd my-project/
```

```
# Update symfony/symfony version  
# in composer.json file
```

```
$ composer update
```

# Adding a new dependency to a Symfony project

# What are Symfony dependencies?

- **Symfony Bundles**

they provide installable features for Symfony applications (e.g. FOSUserBundle, FOSRestBundle)

- **PHP Libraries**

generic PHP packages that don't provide seamless integration with Symfony (e.g. erusev/parsedown, thephpleague/flysystem)

# Adding a bundle to a Symfony project

```
$ cd my-project/  
$ composer require \  
doctrine/doctrine-fixtures-bundle
```

Symfony bundles are **automatically enabled** after installation and some of them provide an **initial configuration**, but they might require to **follow more steps** before using them. Read the **installation instructions** provided by the bundle.

# Adding a library to a Symfony project

```
$ cd my-project/  
$ composer require erusev/parsedown
```

Then, integrate the library into your application by **creating some class or service**.



# Symfony Flex

# What is Symfony Flex?

- It's NOT a new Symfony version.
- It's the new way to **create, manage** and **grow** Symfony applications.
- It **deprecates** the Symfony Installer used until Symfony 3.3 and it's the **default** way of working with Symfony 4.

# What's the purpose of Symfony Flex?

- It **automates** the integration of packages in Symfony applications  
(both bundles and normal libraries/packages)
- It can create dirs and files (config, routes), enable bundles in the kernel, display helpful information, etc.

# How does Symfony Flex work?

- Flex is distributed as a PHP package that must be installed in your application.
- Technically it is a Composer plugin that hooks into the install / update commands to perform advanced tasks.

# Installing Symfony Flex

# Symfony Flex is already installed in new projects

```
$ composer create-project symfony/skeleton my-project
```

- Installing symfony/flex (dev-master 979058c)  
Detected auto-configuration settings for "symfony/flex"  
Setting configuration and copying files
- Installing symfony/security-core (dev-master 87e6003)
- Installing symfony/routing (dev-master fadf939)  
Detected auto-configuration settings for "symfony/routing"
- ...

What's next?

- \* Run your application:
  1. Change to the project directory
  2. Execute the `make serve` command;
  3. Browse to the `http://localhost:8000/` URL.
- \* Read the docs: <https://symfony.com/doc>

# Adding Symfony Flex to an existing project

```
$ cd my-project/  
$ composer require symfony/flex
```

From now on, whenever you install/update a dependency in your project, Symfony Flex will look for the information needed to integrate that package (as explained in detail later).

This is only useful if your project uses the default directory structure proposed for Symfony 4 applications.

# Symfony Flex in practice

# Symfony Flex is a paradigm shift

- **Before:** remove what you don't need.  
The Symfony Standard Edition included most of the Symfony packages (some of them disabled).
- **After:** install what you need.  
The Symfony Skeleton is almost empty and you add dependencies when you need them.

# Installing a Symfony bundle

```
$ composer require acme/mybundle
- Installing acme/mybundle
  Enabling the package as a Symfony bundle
```

If the **type** option in the bundle's **composer.json** is **symfony-bundle**, then Symfony Flex enables the bundle automatically in the app.

# Installing a Symfony feature (e.g. Twig support)

```
$ composer require twig
```

- Installing twig/twig
  - Installing symfony/twig-bridge
  - Installing symfony/twig-bundle
- Detected auto-configuration settings for twig-bundle  
Enabling the package as a Symfony bundle  
Setting configuration and copying files

For other bundles and packages, Symfony Flex is able to perform more advanced tasks (e.g. create a default config file for the bundle). This is only available for packages that define **Symfony Flex recipes**, as explained in the next section.

# Symfony Flex recipes

# Symfony Flex recipes

- A recipe is a set of automated instructions to integrate some package in a Symfony application (e.g. copy some files into the project)
- Recipes contain a `manifest.json` file and, optionally, any number of files and dirs.
- When a package is removed, those changes are reverted automatically too.

# A simple recipe (1 of 2)

```
recipes/
  symfony/
    console/
      4.0/
        manifest.json
        bin/
          console
```

# A simple recipe (2 of 2)

```
// recipes/symfony/console/4.0/manifest.json
{
    "copy-from-recipe": {
        "bin/": "%BIN_DIR%/"
    },
    "aliases": ["cli"]
}
```

# A more complex recipe (1 of 2)

```
recipes/
  symfony/
    web-profiler-bundle/
      4.0/
        manifest.json
        config/
          routing/
            dev/
              web_profiler.yaml
        packages/
          dev/
            web_profiler.yaml
        test/
          web_profiler.yaml
```

# A more complex recipe (2 of 2)

```
// recipes/symfony/web-profiler-bundle/4.0/manifest.json
{
    "bundles": {
        "Symfony\\Bundle\\WebProfilerBundle\\WebProfilerBundle": [
            "dev", "test"
        ],
        "copy-from-recipe": {
            "config/": "%CONFIG_DIR%"
        },
        "aliases": ["profiler", "web-profiler"]
    }
}
```

# Public recipe repositories

- **github.com/symfony/recipes**  
It's the **main** recipe repository. Its contents are curated and carefully reviewed.
- **github.com/symfony/recipes-contrib**  
It's the community repository. Anyone can add recipes for any package.

# Using recipes in your applications

- By default, Symfony Flex **only** uses recipes from the **main repository**.
- If a recipe belongs to the **community repository**, Symfony Flex **asks you for permission** before using it.



# Anatomy of a Symfony 4 project

# Architecture

# Overview of the directory hierarchy

```
<your-project>
  └── config/
  └── bin/
  └── public/
  └── src/
  └── templates/
  └── tests/
  └── var/
  └── vendor/
```

# The config/ directory

```
<your-project>
  └ config/
    ├ bundles.php
    ├ services.yaml
    ├ routes.yaml
    └ packages/
      ├ framework.yaml
      ├ dev/
      ├ prod/
      └ test/
```

The **configuration directory** contains the main config files (bundles.php, services.yaml, routes.yaml) and the config files of each package installed in the app (packages/\*)

# The var/ directory

```
<your-project>
  └ var/
    ├ cache/
    ├ log/
    └ sessions/
```

The **var/** directory contains all generated files such as the cache directory, the recorded logs and the users' sessions.

# The src/ directory

```
<your-project>
  L src/
    └ Kernel.php
    └ Command/
    └ Controller/
    └ Entity/
    └ ...
  L Twig/
```

The **source directory** contains the PHP code of your application and Kernel.php, the file used by Symfony to start your application (it registers routes, services, bundles, etc.)

# The src/ directory in earlier Symfony versions

```
<your-project>
  └ src/
    └ AppBundle/
      ┌ Command/
      ┌ Controller/
      ┌ Entity/
      ┌ ...
      └ Twig/
```

In previous Symfony versions the source code of your application was organized into bundles (e.g. AppBundle).

This is no longer recommended starting from Symfony 3.3.

# The **vendor** directory

```
<your-project>
  └ vendor/
    └ doctrine/
    └ monolog/
    └ sensio/
    └ symfony/
    └ twig/
  └ ...
```

The **vendor** directory contains the dependencies of your project, which are mostly the dependencies of Symfony.

This directory is managed by Composer. Don't change any of its files.

# The public/ directory

```
<your-project>
  └ public/
    ├ index.php
    ├ images/
    ├ css/
    ├ js/
    └ ...
```

**The public directory** contains the publicly accessible files (front controller, web assets, etc.)

This is the **only publicly accessible folder** for Symfony projects.



# Overriding the default directory structure

```
// src/Kernel.php
class Kernel extends BaseKernel
{
    // ...

    public function getCacheDir(): string
    {
        return '/var/cache/my-project';
    }

    public function getLogDir(): string
    {
        return '/var/logs/my-project';
    }
}
```

See [symfony.com/doc/current/configuration/override\\_dir\\_structure.html](https://symfony.com/doc/current/configuration/override_dir_structure.html)

# Configuration

# Symfony 4 configuration

- Configuration formats supported out of the box by Symfony:
  - File based: YAML, XML, PHP.
  - Code-based: annotations
- Format doesn't impact performance
  - All formats are compiled down to PHP before executing the application

# YAML configuration sample

```
# config/packages/framework.yaml
framework:
    secret: '%env(APP_SECRET)%'
    default_locale: en
    #csrf_protection: null
    #http_method_override: true
    #trusted_hosts: null
    #esi: ~
    #fragments: ~
    php_errors:
        log: true
```

# XML configuration sample

```
<!-- config/packages/framework.xml -->
<framework:config
    secret="%env(APP_SECRET)%">
    <framework:csrf-protection />
    <framework:esi />
    <!-- ... -->
</framework>
```

# PHP configuration sample

```
// config/packages/framework.php

$container->loadFromExtension('framework',
array(
    'secret'          => '%env(APP_SECRET)%',
    'default_locale'  => 'en',
    'csrf-protection' => array(),
    // ...
));
```

# PHP Annotations

- They aren't supported in PHP yet  
Other languages support them (Java, C#)
- Beware that they look like comments

**PHP comment**

```
/* ←  
     @Route("...")  
 */
```

**PHP annotation**

```
/** ←  
     @Route("...")  
 */
```

# PHP annotation configuration sample

```
use Symfony\Component\Routing\Annotation\Route;
```

```
class DefaultController
{
    /**
     * @Route("/")
     */
    public function index()
    {
        // ...
    }
}
```

# Summary of configuration formats

	Pros	Cons
<b>Annotations</b>	Easy to read Concise	Commented code No autocompletion Hard to debug
<b>XML</b>	Validation IDE autocompletion	Verbose
<b>YAML</b>	Hierarchical configuration Easy to read	Hard to validate No native PHP support
<b>PHP</b>	Flexible More expressive	No validation

# Best practices for configuration formats

- Use annotations for routing, security, persistence and validation.
- Use YAML/XML for services and configuration options.
- Use PHP if you need a precise control over configuration.

# Environment variables (env vars)

# Configuration based on environment variables

- According to "The Twelve-Factor App" philosophy, config should be strictly separated from code.
- In this context, config is anything that varies between deploys (your local machine, the production server, etc.)

Example: the database credentials.

The Twelve-Factor App: <https://12factor.net>

# Defining environment variables (1/4)

```
# no environment variable  
$ command_name  
  
# temporary environment variable defined  
# only for this command  
$ DB_PASSWORD=1234 command_name
```

# Defining environment variables (2/4)

```
# temporary env variable defined for  
# all the commands executed during  
# this console session  
$ export DB_PASSWORD=1234  
  
# it uses the previous DB_PASSWORD value  
$ command_name
```

# Defining environment variables (3/4)

```
# permanent env variable defined for all the  
# commands executed in this computer  
  
# 1. edit this file  
$ vim ~/.profile  
  
# 2. add this at the end of the file  
export DB_PASSWORD=1234
```

# Defining environment variables (4/4)

```
# permanent env variable defined for all the  
# scripts executed for this website
```

```
# add this in your Apache VirtualHost config  
SetEnv DB_PASSWORD 1234
```



# Defining env vars in Symfony

```
# .env  
APP_ENV=dev  
APP_DEBUG=1  
APP_SECRET=a34cd6c105e568cf416df539b90  
# ...
```

These are the config parameters that may be different on each deployment target of the app (dev, staging, prod, etc.)

The `.env` file is a hidden file at the root of the project.

# Using env vars in config files

```
# config/packages/framework.yaml
framework:
    secret: '%env(APP_SECRET)%'
    default_locale: en
    # ...
```

The special syntax `%env( ... )%` resolves env vars at **runtime**.

# Execution environments

# Developing vs running the application

- When **developing** the application, you need logs and extensive **debug info**.
- When running the application in **production**, you need top **performance**.

# Execution environments

- Symfony allows you to execute the same application with different configuration.
- Each set of configuration values is called **execution environment**.
- Environments are represented by a unique string (**dev, prod, test**).

# The default configuration files

```
<your-project>
  └ config/
    └ packages/
      ┌ my_package.yaml
      ┌ dev/
      |   └ my_package.yaml
      ┌ prod/
      |   └ my_package.yaml
      └ test/
        └ my_package.yaml
```

Packages can define a **different configuration** depending on the **execution environment**.

# Front controller selects the environment

```
// public/index.php  
$kernel = new Kernel(  
    getenv('APP_ENV'),  
    getenv('APP_DEBUG'))  
;
```

To execute the application in different environments and to enable/disable debugging, change the value of those **APP\_\*** environment variables in the **.env** file.



# Which configuration file is loaded by Symfony?

```
// src/Kernel.php

class Kernel extends BaseKernel

{
    // ...

    protected function configureContainer(
        ContainerBuilder $container, LoaderInterface $loader)

    protected function configureRoutes(
        RouteCollectionBuilder $routes);

}
```

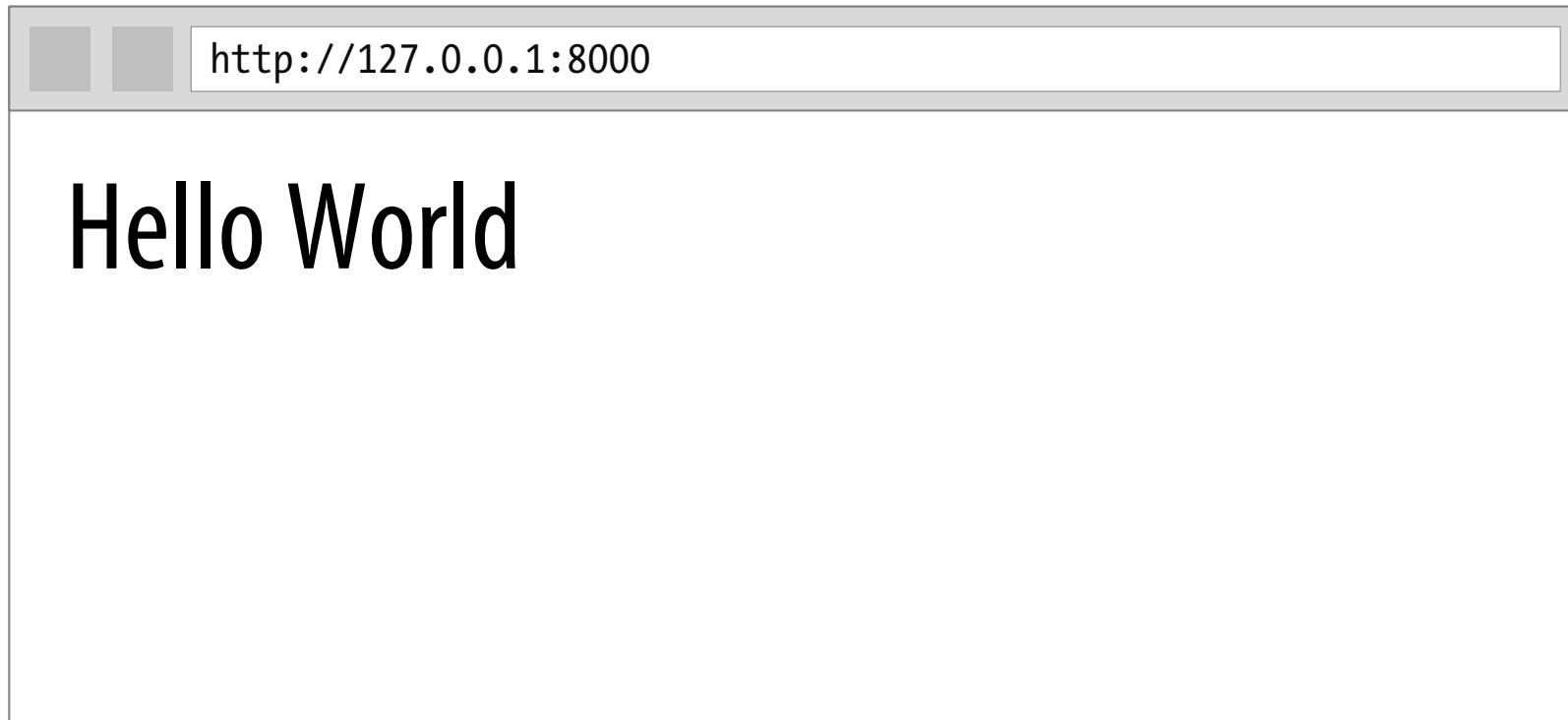


# Hello Symfony World

# Building a Hello World application

# Hello World Application

Let's build the simplest application to show how does Symfony work.



# HTTP under the hood

GET / HTTP/1.1

Host: 127.0.0.1:8000

User-Agent: Mozilla/5.0 Firefox

Accept: text/html,application/xhtml+xml;q=0.9,\*/\*;q=0.8

Accept-Language: en;q=0.8,es;q=0.3,fr;q=0.2

Accept-Encoding: gzip, deflate

Cache-Control: max-age=0

**HTTP Request**

sent by the browser

HTTP/1.1 200 OK

Host: 127.0.0.1:8000

Cache-Control: no-cache

Date: Thu, 14 Aug 201X 15:12:19 GMT

Content-Type: text/html; charset=UTF-8

X-Debug-Token: 1dd824

X-Debug-Token-Link: /\_profiler/1dd824

**HTTP Response**

received from the server

Hello World

# Processing HTTP requests with raw PHP code

```
<?php

// load and initialize global libraries
require_once 'model.php';
require_once 'controllers.php';

$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' == $uri) {
    list_action();
} elseif ('/index.php/show' == $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

## CAUTION

Extremely hard to maintain  
and error prone code.

# Sending HTTP responses with raw PHP code

```
<?php  
  
$link = mysql_connect('localhost', 'myuser', 'mypassword');  
mysql_select_db('blog_db', $link);  
  
$result = mysql_query('SELECT id, title FROM post', $link);  
  
$posts = array();  
while ($row = mysql_fetch_assoc($result)) {  
    $posts[] = $row;  
}  
  
mysql_close($link);  
  
// include the HTML+PHP template  
require 'templates/list.php';
```

## CAUTION

Extremely hard to maintain  
and error prone code.



# HTTP requests and responses in Symfony

```
// src/Controller/DefaultController.php
namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController
{
    /**
     * @Route("/")
     */
    public function hello()
    {
        return new Response('Hello World');
    }
}
```

This code shows **Hello World** when browsing the homepage of the site.  
Run "**composer require annotations**" to add support for annotations.

# Web server configuration

## Best-practice

Use the PHP built-in web server  
when developing Symfony  
applications locally.

# Using the PHP built-in web server (1/2)

```
$ cd my-project/  
$ php -S 127.0.0.1:8000 \  
-t public/
```



# Using the PHP built-in web server (2/2)

```
$ cd my-project/  
$ composer require server  
$ ./bin/console server:start
```

```
[OK] Server listening on http://127.0.0.1:8000
```

# Using Apache Web Server

public/ is the only  
public directory for  
Symfony applications

```
<VirtualHost *:80>
```

```
    ServerName      my-project.dev
    DocumentRoot   "/projects/my-project/public"
    DirectoryIndex index.php
```

```
<Directory "/projects/my-project/public">
```

```
    AllowOverride None
    Allow from All
</Directory>
```

```
<IfModule mod_rewrite.c>
```

```
    RewriteEngine On
    RewriteCond  %{REQUEST_FILENAME} !-f
    RewriteRule  ^(.*)$ index.php [QSA,L]
```

```
</IfModule>
```

```
</VirtualHost>
```

# Using Nginx Web Server

public/ is the only  
public directory for  
Symfony applications

```
server {  
    server_name my-project.dev;  
    root /projects/my-project/public; <-----  
  
    location / {  
        try_files $uri /index.php$is_args$args;  
    }  
  
    error_log /var/log/nginx/project_error.log;  
    access_log /var/log/nginx/project_access.log;  
}
```

# Setting up permissions

# Understanding the permission issue

Web server

user: [www-data](#)

Reads and writes to  
→

Command console

user: [johnsmith](#)

Reads and writes to  
→

<your-project>

└ config/	<a href="#">READ</a>
└ public/	<a href="#">READ</a>
└ src/	<a href="#">READ</a>
└ var/	<a href="#">READ</a>
└ cache/	<a href="#">READ</a> <a href="#">WRITE</a>
└ log/	<a href="#">READ</a> <a href="#">WRITE</a>
└ vendor/	<a href="#">READ</a>

# Solving the permission issue

- In your **local machine** you don't need to do anything: Symfony solves the problem with the `umask()` PHP function.
- In **production servers**, you must run the `cache:warmup` command to generate all the files needed during runtime.

More info: [symfony.com/doc/current/setup/file\\_permissions.html](https://symfony.com/doc/current/setup/file_permissions.html)

# Permissions of the console script

```
#!/usr/bin/env php
use App\Kernel;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Dotenv\Dotenv;
use Symfony\Component\Console\Input\ArgvInput;
use Symfony\Component\Debug\Debug;

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

// ...

if ($debug) {
    umask(0000);
    // ...
}
```

# Permissions of the index.php controller

```
use App\Kernel;
use Symfony\Component\Dotenv\Dotenv;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Debug\Debug;

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

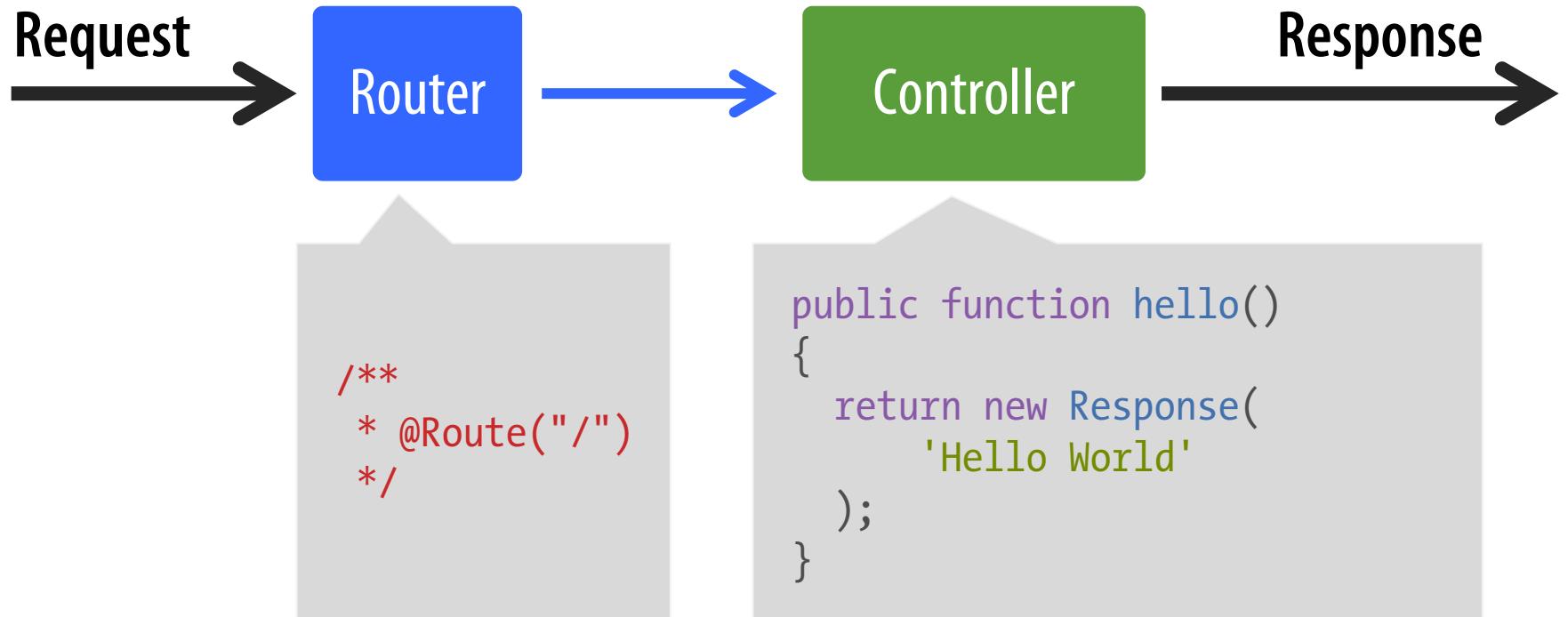
if (getenv('APP_DEBUG')) {
    umask(0000);

    // ...
}

// ...
```

# The Request - Response flow

# The simplest Request-Response Flow



# Symfony is ...

- ✓ A Request/Response framework.
- ✓ An HTTP framework.
- ✗ A MVC framework.  
(Model-View-Controller)



# Rendering a template (1 of 2)

```
// src/Controller/DefaultController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends AbstractController
{
    /**
     * @Route("/")
     */
    public function hello()
    {
        return new Response('Hello World');
        return $this->render('index.html.twig');
    }
}
```

Twig is a templating format which will be explained later.



# Rendering a template (1 of 2)

```
// src/Controller/DefaultController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends AbstractController
{
    /**
     * @Route("/")
     */
    public function hello()
    {
        return new Response('Hello World');
        return $this->render('index.html.twig');
    }
}
```



**Using the base  
AbstractController is  
optional, but it provides  
lots of useful shortcuts.**



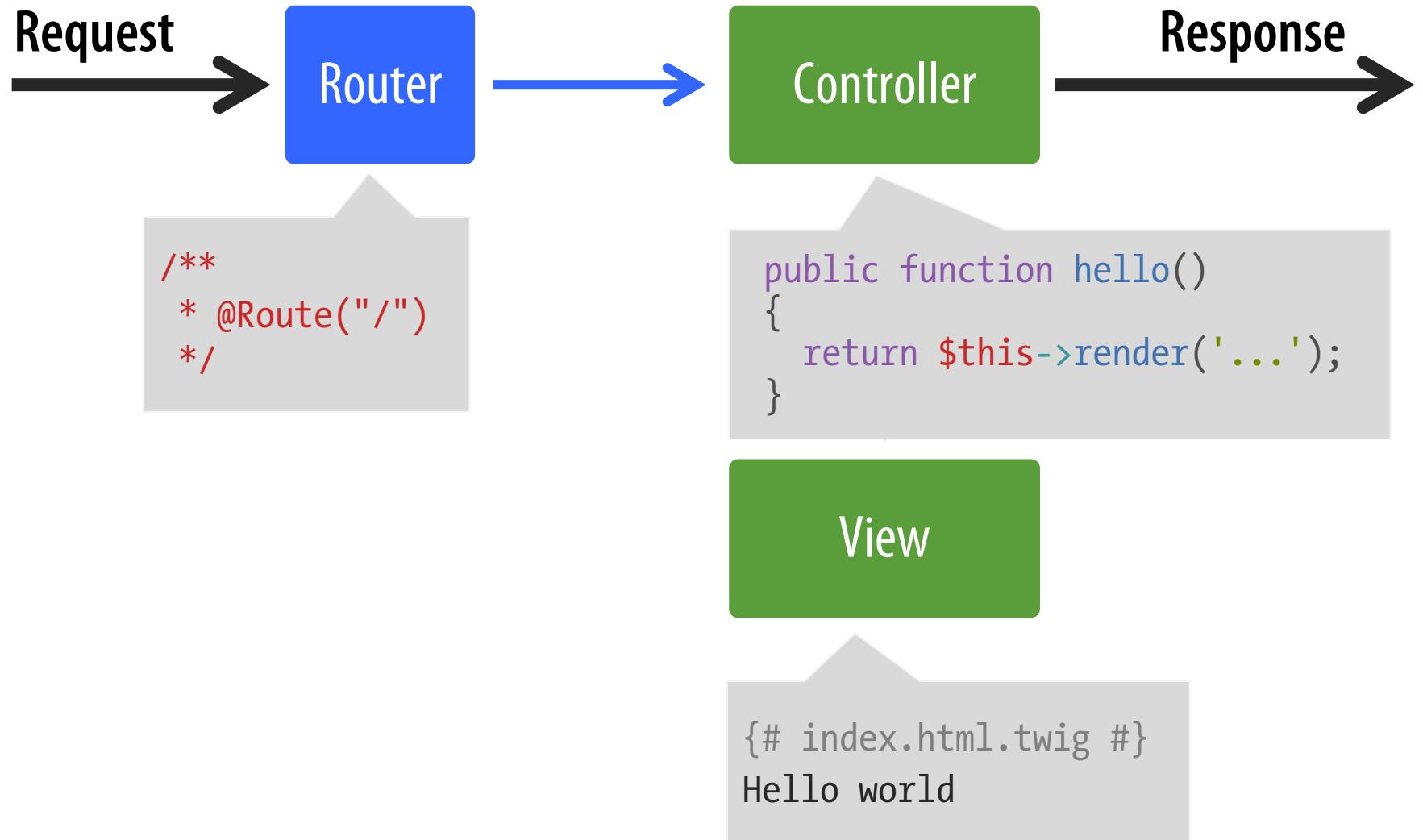
# Rendering a template (2 of 2)

```
{# templates/index.html.twig #}
```

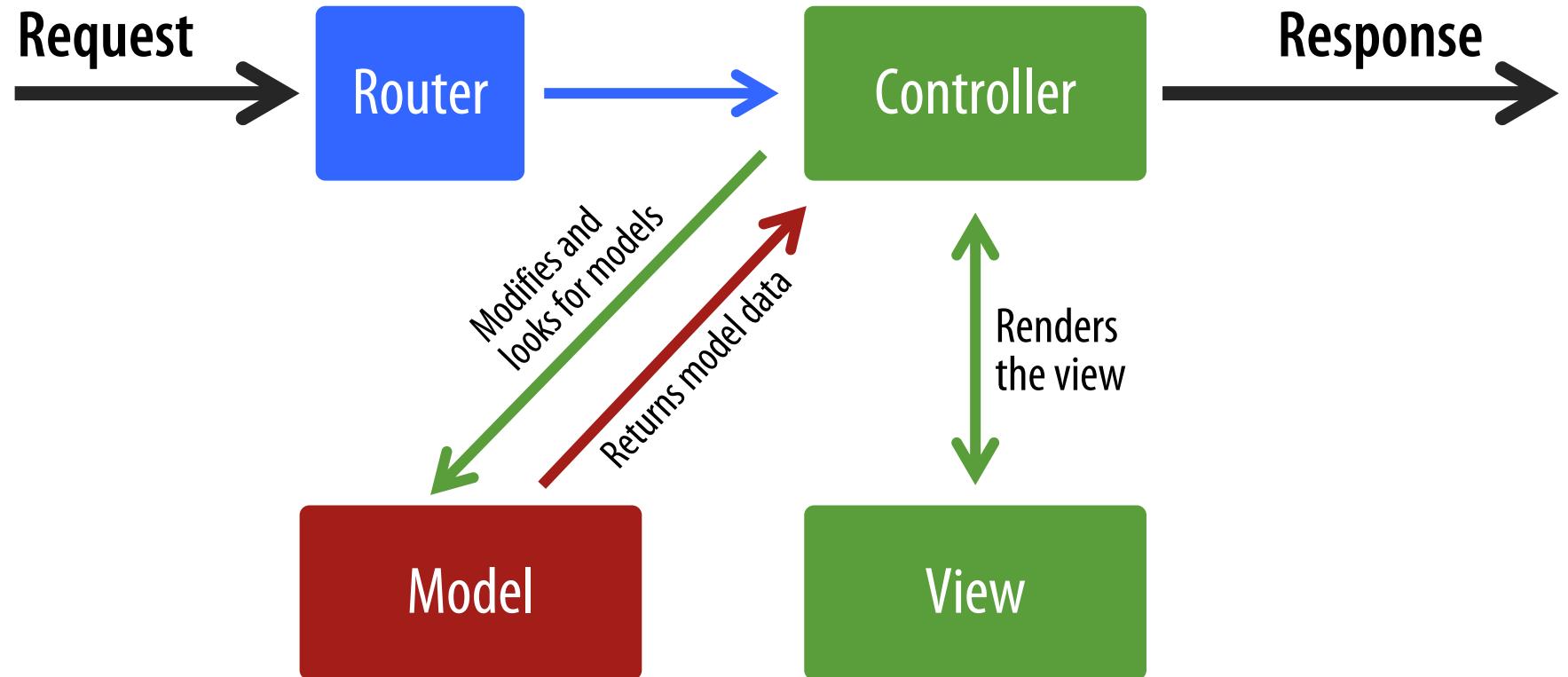
```
Hello world
```

Install Twig support in your project (run "**composer require twig**") before rendering Twig templates.

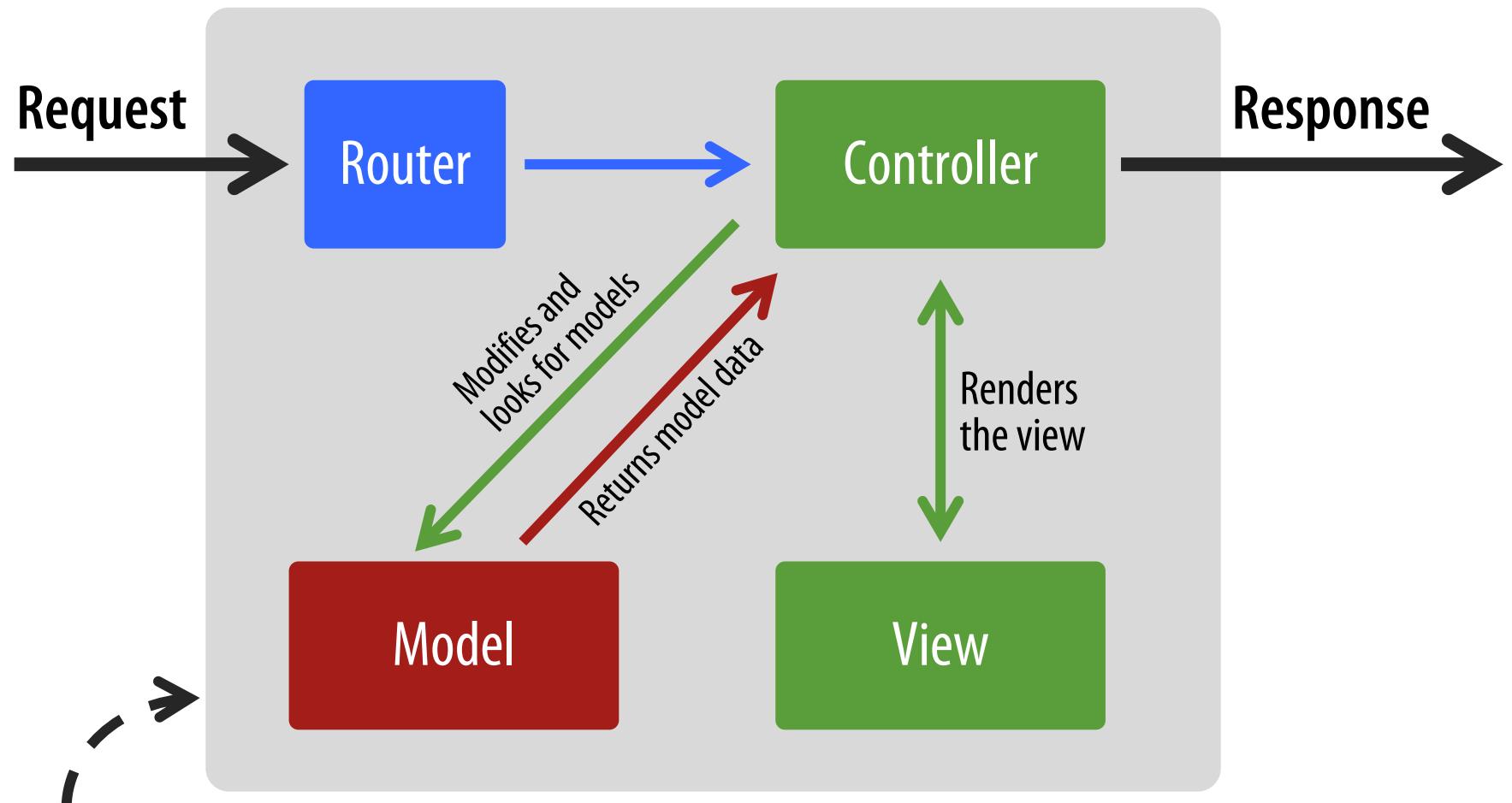
# The advanced Request-Response Flow



# The complete Request-Response Flow

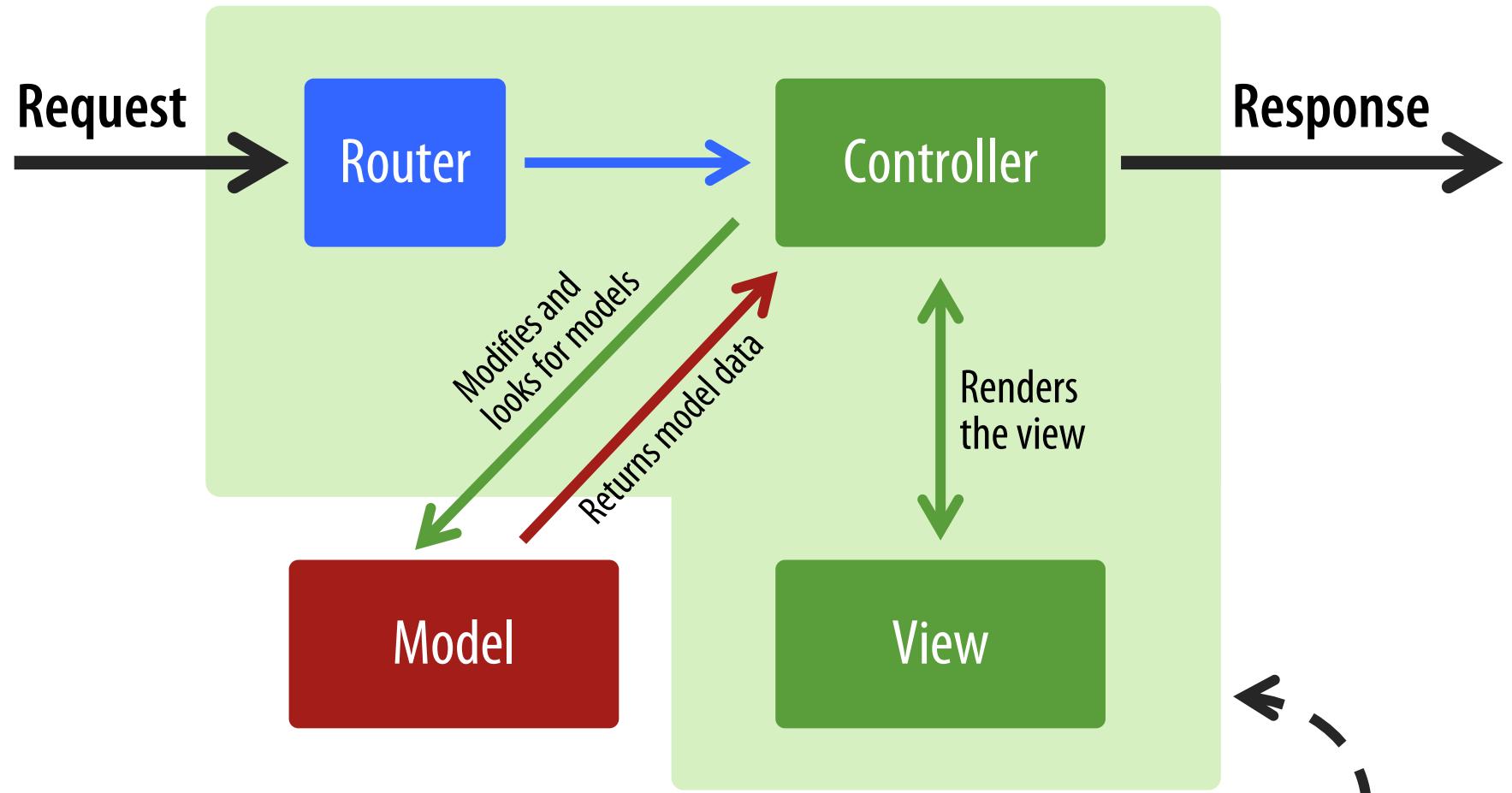


# The complete Request-Response Flow



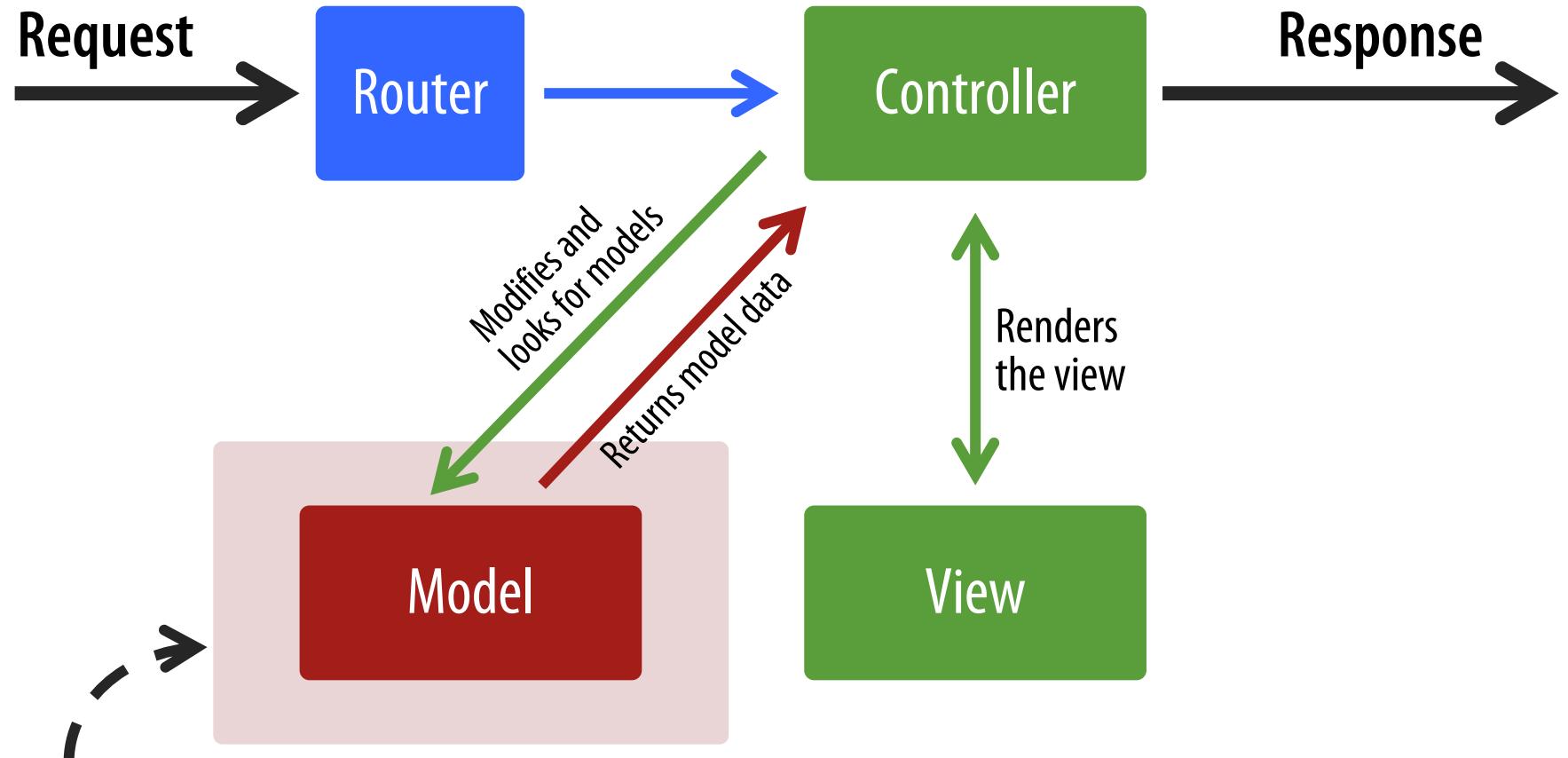
Your entire  
Symfony project

# The complete Request-Response Flow



This is what  
Symfony provides

# The complete Request-Response Flow



**This is not part of Symfony**  
(use Doctrine, PDO or your own system)

# The Routing component

# The Routing component

- It associates URLs with controllers, so Symfony knows the code to execute to respond to requests.
- It generates URLs so links displayed on templates are always valid even when the structure of the application changes.

# The Routing configuration

- It can be defined in any format: YAML, XML, PHP or annotations.
- Annotations are recommended because it puts routes + controllers in the same file.
- **YAML** was common a few years ago.
- **XML** is too verbose, **PHP** is too low level.

# Some annotations require a special package

```
$ cd your-project/  
$ composer require annotations
```

# A simple route example

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog", name="blog_list")
     */
    public function list()
    {
        // ...
    }
}
```

# A route with placeholders (variables)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list")
     */
    public function list($page)
    {
        // $page variable is available here
        // ...
    }
}
```

# A route with default values (1 of 3)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list", defaults={"page": "1"})
     */
    public function list($page)
    {
        // ...
    }
}
```

# A route with default values (2 of 3)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page?1}", name="blog_list")
     */
    public function list($page)
    {
        // ...
    }
}
```

requires  
**Symfony 4.1**

# A route with default values (3 of 3)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list")
     */
    public function list($page = 1)
    {
        // ...
    }
}
```

# A route with constraints (1 of 2)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list",
     *       requirements={"page": "\d+"})
     */
    public function list($page)
    {
        // ...
    }
}
```

# A route with constraints (2 of 2)

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;  
use Symfony\Component\Routing\Annotation\Route;  
  
class BlogController extends AbstractController  
{  
    /**  
     * @Route("/blog/{page<\d+>}", name="blog_list")  
     */  
    public function list($page)  
    {  
        // ...  
    }  
}
```

requires  
**Symfony 4.1**

# Restricting the request HTTP method

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list", methods={"GET"})
     */
    public function list($page)
    {
        // ...
    }
}
```

# A complex route example (annotations)

```
use Symfony\Component\Routing\Annotation\Route;

/**
 * @Route("/blog/{page<\d+>}?1", name="blog_list")
 *      methods={ "GET", "HEAD" })
 */

public function list($page)
{
    // ...
}
```

# A complex route example (YAML)

```
# config/routes.yaml
blog_list:
    path: /blog/{page}
    controller: App\Controller\DefaultController::index
    defaults:
        page: 1
    requirements:
        page: \d+
    methods: [GET, HEAD]
```



# Introduction to Twig

# What is Twig

# What is Twig

Twig is a modern template engine for PHP.

The official website for the project is  
[twig.symfony.com](http://twig.symfony.com)



This is **the official logo** of the project

In English, **twig** literally means  
«A small thin branch of a tree or bush»

# Twig features

- **Fast**

Templates are compiled to raw PHP before executing them

- **Secure**

By default, contents are escaped before displaying them. It also includes a sandbox mode to restrict template execution

- **Modern**

Template-oriented syntax, concise, flexible and full-featured for modern web application

# Twig is more concise than PHP

```
{ { variable } }
```

Twig is **secure by default** because it escapes contents before displaying them.

```
<?php  
echo htmlspecialchars(  
    $variable,  
    ENT_QUOTES,  
    'UTF-8'  
)  
?>
```

Secure PHP code is much more verbose.

# Twig's template oriented syntax

```
{% for user in users %}
```

```
    * {{ user.name }}
```

```
{% else %}
```

No users have been found.

```
{% endfor %}
```

for ... else is a convenient  
construct provided by Twig and  
which doesn't exist in PHP

# Basic syntax

# Concise syntax

{# ... comment something ... #}

{% ... do something ... %}

{{ ... display something ... }}

These are the three special tags used to separate Twig code from regular template contents.

# Rendering variables

# Abstracting access to variables

```
{ { article.title } }
```

Twig templates use the "dot syntax" to access properties from PHP objects and associative arrays.

# Abstracting access to variables

```
echo $article['title'];
echo $article->title;
echo $article->title();
echo $article->getTitle();
echo $article->isTitle();
echo $article->hasTitle();
```

When using {{ article.title }} in a template, Twig will look for these keys/properties/methods and in this order.

# Strict variables

```
# config/packages/twig.yaml  
twig:  
    strict_variables: false  
  
{{ article.title }}
```

**Fails silently** when the variable doesn't exist (page shows a blank spot)

```
# config/packages/twig.yaml  
twig:  
    strict_variables: true  
  
{{ article.title }}
```

**Throws an exception** when the variable doesn't exist.

# Filters and functions

# Filters format contents

```
{% post.publishedAt|date('d/m/Y') %}
```

```
{% post.title|lower %}
```

```
{% post.title|upper %}
```

```
{% post.title|capitalize %}
```

```
{% post.title|title %}
```

```
{% post.tags|sort|join(', ') %}
```

```
{% post.author|default('Anonymous') %}
```

# Built-in filters

- abs
- batch
- capitalize
- convert\_encoding
- date
- date\_modify
- default
- escape
- first
- format
- join
- json\_encode
- keys
- last
- length
- lower
- merge
- nl2br
- number\_format
- raw
- replace
- reverse
- round
- slice
- sort
- split
- striptags
- title
- trim
- upper
- url\_encode

# Functions generate contents

Hi {{ random(['John', 'Tom', 'Paul']) }}!

```
{% for i in range(0, 10, 2) %}
    {{ cycle(['odd', 'even'], i) }} <br/>
{% endfor %}
```

# Built-in functions

- attribute
- date
- min
- block
- dump
- parent
- constant
- include
- random
- cycle
- max
- range
- template\_from\_string
- source

# Output escaping

# Automatic output escaping

Hi {{ name }} !

The variable name is automatically escaped if it contains a string

# Automatic output escaping

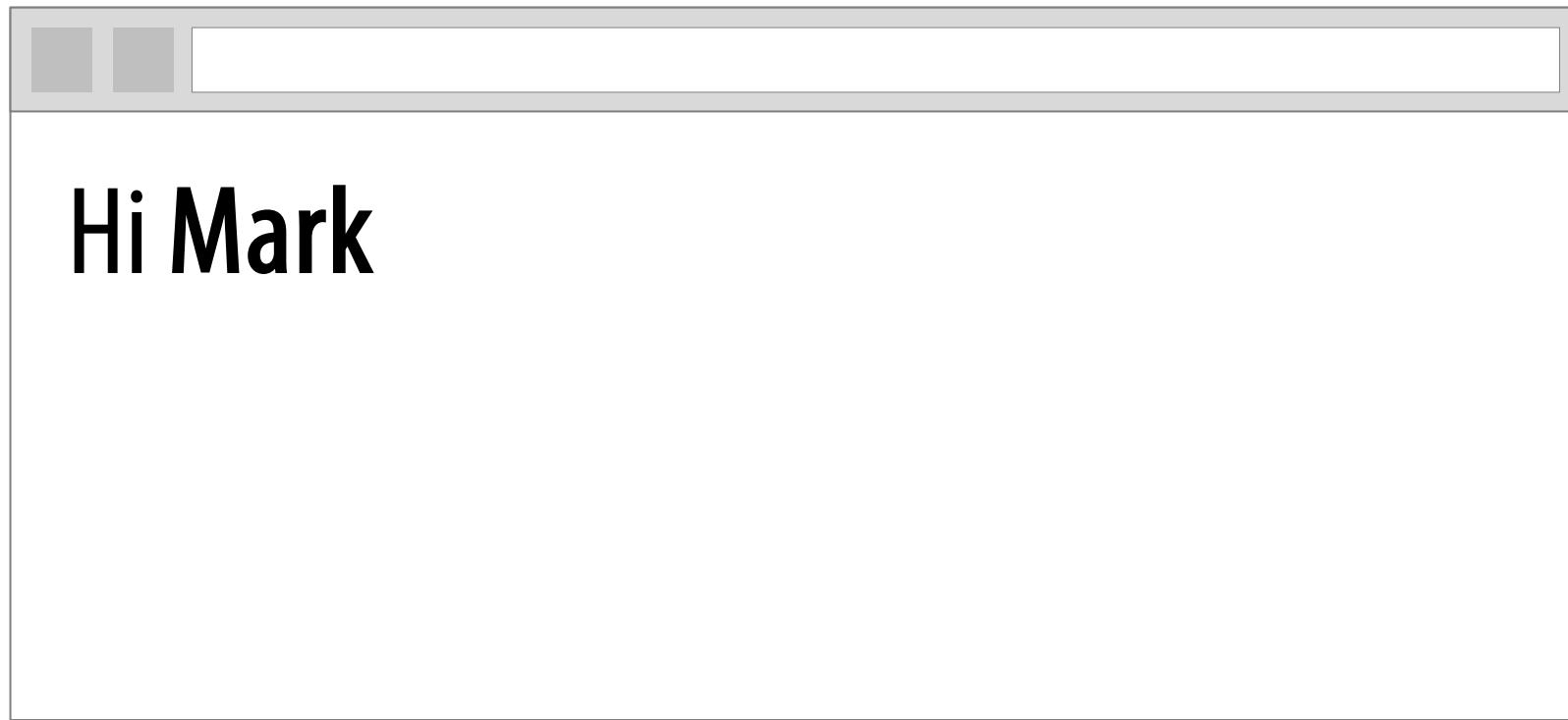
Hi {{ name }}

```
$name = '<strong>John</strong>';
```

# Automatic output escaping

Expected output

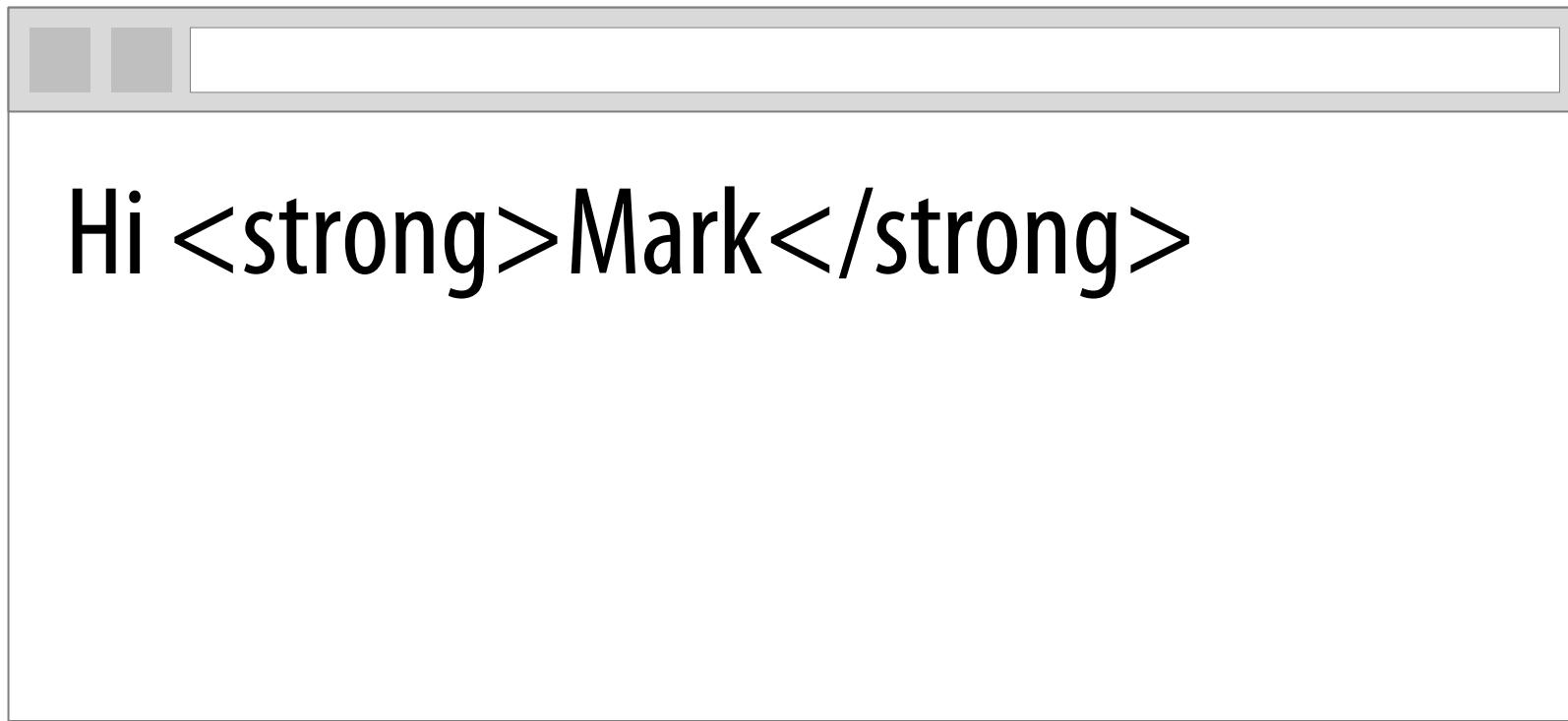
Hi <strong>Mark</strong>



# Automatic output escaping

Real output

Hi &lt;strong&gt;Mark&lt;/strong&gt;



# Control structures

# Comparison of control structures

## Twig

if                  for

elseif

else

## PHP

break              for  
continue        foreach  
do ... while    goto  
if                switch  
elseif            while  
else

# Making decisions

```
{% if product.stock > 10 %}
```

Available

```
{% elseif product.stock > 0 %}
```

Only {{ product.stock }} left!

```
{% else %}
```

Sold-out!

```
{% endif %}
```

# Iterating over a collection

```
{% for post in posts if post.active %}  
    <h2>{{ post.title }}</h2>  
    {{ post.body }}  
{% else %}  
    No published posts yet.  
{% endfor %}
```

The **if** statement is executed on each iteration,  
allowing you to conditionnally show collection items.

# The loop context

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.index0	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration
loop.last	True if last iteration
loop.length	The number of items in the sequence
loop.parent	The parent context

# Operators

# Basic operators

## Mathematical

+ - \* / \*\* // %

## Logical

and or not ( ... )

b-and b-xor b-or

# Comparison operators

== != < > <= >=

starts with ends with matches

{% if url starts with 'https://' %}

{% if fileName ends with '.txt' %}

{% if phone matches '/^[\d\.]+\$/ %}

# Concatenation operator

~

```
{{ 'Hello' ~ user.fullName ~ '!' }}
```

```
{{ firstName ~ ' ' ~ lastName }}
```

# Interpolation operator

#{ }

```
{{ 'Hello #{ user.name }!' }}
```

```
{{ 'Discount: #{ product.price *  
discount / 100 }' }}
```

# Containment operator

in      not in

```
{% if name not in user.friends %}
```

    Add as a friend

```
{% endif %}
```

```
{% if login in password %}
```

    ERROR password can't contain login!

```
{% endif %}
```

# Other operators

is      is not

```
{% if number is odd %}  
{% if number is not  
    divisible by(3) %}
```

# Built-in tests

```
{% if numElements is constant('Object::CONSTANT') %}  
{% if user.login is defined %}  
{% if user.friends|length is divisible by(3) %}  
{% if user.cart is empty %}  
{% if product.photos|length is even %}  
{% if product.photos|length is odd %}  
{% if user.badges is iterable %}  
{% if user is null %}  
{% if user is same as(logged_user) %}
```

Check out the official Twig reference at <http://twig.sensiolabs.org/documentation>

# Other operators

..

```
{% if number in 1..10 %}
```

```
{% for letter in 'a'..'z' %}
```

Equivalent to PHP `range()` function, but more concise.

# Other operators

?    ?:    ??

```
{{ article.published ? 'yes' : 'no' }}
```

```
{{ article.author ?: 'Anonymous' }}
```

```
<div class="{{ category == 'index' ? 'active' }}">
```

```
{{ num_items ?? 0 }}
```

# Whitespace control

# Whitespace control

```
{% spaceless %} \n
• <p> \n
.. Hello <strong>{{ name }}</strong>! \n
• </p> \n
{% endspaceless %}
```

Output

```
<p> \n
.. Hello <strong>Hugo</strong>! \n
• </p>
```

# Whitespace control

```
<p>
  Hello <strong> {{- name }} </strong>!
</p>
```

```
<p>Hello <strong>Hugo </strong>!</p>
```

# Whitespace control in practice

```
<ul>  
  {% for i in 1..3 %}  
    <li>{{ i }}</li>  
  {% endfor %}  
</ul>
```

```
<ul>  
  <li>1</li>  
  <li>2</li>  
  <li>3</li>  
</ul>
```

# Whitespace control in practice

```
<ul>  
    {% for i in 1..3 %}  
        <li>{{ i }}</li>  
    {% endfor %}  
</ul>
```

```
<ul>  
    <li>1</li>  
    <li>2</li>  
    <li>3</li>  
</ul>
```

# Whitespace control in practice

```
<ul>  
    {%- for i in 1..3 %}  
        <li>{{ i }}</li>  
    {% endfor %}  
</ul>
```

```
<ul>      <li>1</li>  
          <li>2</li>  
          <li>3</li>  
</ul>
```

# Whitespace control in practice

```
<ul>  
    {%- for i in 1..3 -%}  
        <li>{{ i }}</li>  
    {% endfor %}  
</ul>
```

```
<ul><li>1</li>  
    <li>2</li>  
    <li>3</li>  
</ul>
```

# Whitespace control in practice

```
<ul>
    {%- for i in 1..3 -%}
        <li>{{ i }}</li>
    {%- endfor -%}
</ul>
```

```
<ul><li>1</li><li>2</li><li>3</li></ul>
```

# Whitespace control in practice

```
{% spaceless %}  
<ul>  
    {% for i in 1..3 %}  
        <li>{{ i }}</li>  
    {% endfor %}  
</ul>  
{% endspaceless %}
```

```
<ul><li>1</li><li>2</li><li>3</li></ul>
```

# Best practice

- Ignore white space control unless it's critical for some template.
- Always make template readable and formatted for humans, not machines.

# Template inclusion

# Template inclusion

The `include()` function evaluates a template and returns the generated contents.

```
<header>
  {{ include('menu.html.twig') }}
</header>
```

# Template inclusion

The included template can be stored anywhere in your application:

```
<header>
{{ include('common/menu.html.twig') }}
</header>
```

# Variable scope

- Included templates can access to all the parent template's variables.
- Use `with_context` option to control this.

```
<header>
{{ include('common/menu.html.twig',
           with_context = false) }}
</header>
```

# Passing new variables or renaming them

```
<header>
```

```
  {{ include(  
    'common/menu.html.twig',  
    { var1: '...', var2: '...' },  
    with_context = false  
  ) }}  
</header>
```

# Template inheritance

# The need of template inheritance

- In a given website, most of its pages share **the same structure**.
- Using the **include()** function is possible, but **inefficient**.
- Template **inheritance** is the best way to solve this problem.

# Creating the parent template

Contains all the common HTML elements shared by all pages and defines the blocks of contents that can be filled in by child templates.

```
{# templates/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>My website</title>
    </head>
    <body>
        <h1>My Symfony Application</h1>
        {% block body %}{% endblock %}
    </body>
</html>
```

# Creating the child template

```
{% extends 'base.html.twig' %}
```

```
{% block body %}  
    <h2>Latest posts</h2>  
    {{ include('posts.twig') }}  
{% endblock %}
```

# Extending from the parent template

```
{% extends 'base.html.twig' %}
```

- It must be the **first instruction** of the template.
- A template can only inherit from one template.
- There is no **inheritance level** limit (parent, child, grandchild, etc.)

# Filling the parent's blocks

```
{% block body %}  
    <h2>Latest posts</h2>  
    {{ include('posts.twig') }}  
{% endblock body %}
```

- Child templates **can** fill-in the blocks defined in the parents, but it's **not mandatory** to do it.
- Child templates cannot add **content outside a block** element. Otherwise, Twig will show an error.
- Inside a **block** content you can use any Twig element, including expressions and include() function.

# Parent templates usually define lots of blocks

```
{# templates/base.html.twig #}
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8">
        <title>{% block title %}{% endblock %}</title>
    </head>

    <body id="{% block body_id %}{% endblock %}">
        <h1>My Symfony Application</h1>
        {% block body %}{% endblock %}
    </body>

</html>
```

# Child templates usually fill most of the blocks

```
{% extends 'base.html.twig' %}
```

```
{% block body_id %}blog_index{% endblock %}  
{% block title %}Latest Posts{% endblock %}
```

```
{% block body %}  
    <h2>Latest posts</h2>  
    {{ include('posts.twig') }}  
{% endblock body %}
```

# Alternative notation for short blocks

```
{% extends 'base.html.twig' %}
```

```
{% block body_id 'blog_index' %}
```

```
{% block title 'Latest Posts' %}
```

```
{% block body %}
```

```
    <h2>Latest posts</h2>
```

```
    {{ include('posts.twig') }}
```

```
{% endblock body %}
```

# Reusing the content of any block

```
{% extends 'base.html.twig' %}
```

```
{% block body_id 'blog_index' %}
```

```
{% block title 'Latest Posts' %}
```

```
{% block body %}
```

```
    <h2>{{ block('title') }}</h2>
```

```
    {{ include('posts.twig') }}
```

```
{% endblock body %}
```

# Parent templates can define default contents

```
{# templates/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>
            {% block title %}My application{% endblock %}
        </title>
    </head>

    <body id="{% block body_id %}{% endblock %}">
        <h1>My Symfony Application</h1>
        {% block body %}{% endblock %}
    </body>
</html>
```

# Default contents in child templates

```
{% extends 'base.html.twig' %}
```

```
{% block title %}  
{% endblock %}
```

Removes  
the default parent value

```
{% block title %}  
    Blog  
{% endblock %}
```

Overrides  
the default parent value

```
{% block title %}  
    Blog - {{ parent() }}  
{% endblock %}
```

Modifies  
the default parent value

# Macros

# What is a Twig macro

- Macros are comparable with **functions** in regular programming languages.
- They are useful to put often used **HTML idioms** into reusable elements to **not repeat yourself**.
- They must be **imported** before using them.

# Defining a macro

```
{% macro input(name, value, type='text', size=20) %}  
  <input type="{{ type }}"  
        name="{{ name }}"  
        value="{{ value|e }}"  
        size="{{ size }}" />  
{% endmacro %}
```

# Using a macro defined in an external file

```
{% import "form_macros.html.twig" as utils %}

<form>
  {{ utils.input('username') }}
  {{ utils.input('password', null, 'password') }}
</form>
```

# Using a macro defined in the same file

```
{% macro input(name, value, type = 'text', size = 20) %}  
  <input type="{{ type }}" name="{{ name }}"  
        value="{{ value|e }}" size="{{ size }}" />  
{% endmacro %}
```

```
{% import _self as utils %}
```

```
<form>  
  {{ utils.input('username') }}  
  {{ utils.input('password', null, 'password') }}  
</form>
```

# Debug

# Accurate error messages

```
{{ rand(['A', 'B', 'C', 'D']) }}!
```

## Twig\_Error\_Syntax

The function "rand" does not exist. Did you mean  
"random" in "hello.twig" at line 3

# Dumping variables

```
{% set names      = ['John', 'Tom', 'Paul'] %}  
{% set numbers = 1..5 %}
```

```
{{ dump(names) }}
```

```
{{ dump(names, numbers) }}
```

```
 {{ dump() }}
```

dumps all variables  
defined in the template

First, install debug support in your app (run "composer require debug")

# PHP compilation

# PHP compilation process

- To increase **performance**, Twig templates are compiled down to PHP.
- The **impact** on performance over raw PHP templates is **negligible**.
- In **development**, changed templates are recompiled. Not in **production**.

# A simple Twig template

```
{# A comment #}
```

```
Hello {{ name }}!
```

# The resulting PHP compiled template

```
/* index.html.twig */
class _TwigTemplate_d2793ba4e21454af9bfe3bc75aaa83b5324a893143a805c121808f3902a38ca6
extends Twig_Template {
    public function __construct(Twig_Environment $env) { ... }

    protected function doDisplay($context, $blocks = array()) {
        // line 2
        echo "Hello ";
        echo twig_escape_filter($this->env, (isset($context["name"]) ?
$context["name"] : $this->getContext($context, "name")), "html", null,
true);
        echo "!";
    }

    // ...
}
```



# Twig & Symfony integration

# Adding Twig support in Symfony projects

# Adding Twig support in Symfony projects

```
$ cd my-project/  
$ composer require twig
```

# Global variables

# Defining global variables

```
# config/packages/twig.yaml
twig:
    # ...
    globals:
        ga_tracking: "UA-xxxxx-x"
        site_version: "v3.1"
```

Global variables are automatically injected into every Twig template of the application.

# Using global variables

```
<head>  
  <meta name="version"  
        content="{{ site_version }}">  
  ...  
</head>
```

Global variables are used as any other regular variable.  
The only difference is that they are always available.

# Global objects

# Global objects

`{{ app.request }}`

`{{ app.session }}`

`{{ app.user }}`

Symfony provides you with the `app` global variable that includes shortcuts to the `user`, the `session` and the `request` objects.

# URLs and links

# Generating URLs and links

Relative URL

```
<a href="{{ path('homepage') }}">  
    Back to Home  
</a>  
<a href="/">Back to home</a>
```

Absolute URL

```
<a href="{{ url('homepage') }}">  
    Back to Home  
</a>  
<a href="http://example.com">Back to Home</a>
```

# Web assets

# Add support for assets in Symfony apps

```
$ cd my-project/  
$ composer require asset
```

# Linking to web assets stored in public/ directory

```

```

```
<link rel="stylesheet"  
      href="{{ asset('css/blog.css') }}"  
      type="text/css" />
```

Assets must be located in the `public/` directory.

# Installing web assets

```
$ php bin/console  
assets:install --symlink
```

This command copies/symlinks bundle's assets to **public/**, so they can be accessed by **asset()** function.

# Defining asset version

```
# config/packages/framework.yaml
framework:
    # ...
assets:
    version: "v=2"
```

```

```

Template

```

```

Output

# Defining asset base URL

```
# config/packages/framework.yaml
framework:
    # ...
assets:
    base_urls:
        - 'http://static.example.com'
```

```

```

Template

```

```

Output

# Versions based on asset contents

- It requires using front-end tools like Webpack.
- Symfony supports this kind of versioning via Webpack Encore and manifest files.
- [symfony.com/doc/current/frontend/encore/versioning.html](https://symfony.com/doc/current/frontend/encore/versioning.html)

# Filters and functions for controllers

# Controller functions

```
{{ render('http://...') }}
```

```
{{ render(controller(
    'App\\Controller\\ArticleController::latest', { max: 3 }
)) }}
```

```
{{ render_esi(controller(
    'App\\Controller\\ArticleController::latest', { max: 3 }
)) }}
```

# Filters and functions for forms

# Add support for forms in Symfony apps

```
$ cd my-project/  
$ composer require form
```

# Form functions

```
{% form_theme form 'form/fields.html.twig' %}

{{ form_start(form) }}
{{ form_errors(form) }}

<div>
    {{ form_label(form.task) }}
    {{ form_errors(form.task) }}
    {{ form_widget(form.task) }}
</div>

<div>
    {{ form_widget(form.save) }}
</div>

{{ form_end(form) }}
```

Documentation and examples:

[symfony.com/doc/current/reference/twig\\_reference.html](https://symfony.com/doc/current/reference/twig_reference.html)

# Filters and functions for security

# Add support for security in Symfony apps

```
$ cd my-project/  
$ composer require security
```

# Security functions

```
{% if is_granted('ROLE_ADMIN') %}  
    <a href="...">Delete</a>  
{% endif %}  
  
<a href="{{ logout_path() }}">  
    Close session  
</a>  
<a href="{{ logout_url() }}">  
    Close session  
</a>
```

Documentation and examples:  
[symfony.com/doc/current/reference/twig\\_reference.html](https://symfony.com/doc/current/reference/twig_reference.html)

# Filters and functions for translation

# Add support for translations in Symfony apps

```
$ cd my-project/  
$ composer require translation
```

# Translation filters

```
{{ message|trans }}
```

```
{{ message|transchoice(5) }}
```

```
{{ message|trans(
    {'%name%': 'John'}, "app" ) }}
```

```
{{ message|transchoice(5,
    {'%name%': 'John'}, 'app' ) }}
```

# Commands

# Twig linter

Checks if the syntax of the Twig templates is valid.

```
$ php bin/console lint:twig path/
```

```
OK in templates/blog/show.html.twig
OK in templates/comment/_comments.html.twig
OK in templates/comment/_form.html.twig
OK in templates/comment/new.html.twig
OK in templates/layout.html.twig
OK in templates/page/_sidebar.html.twig
OK in templates/page/about.html.twig
OK in templates/page/contact.html.twig
OK in templates/page/contact_email.txt.twig
OK in templates/page/index.html.twig
```

```
10 / 10 valid files
```

# Twig debugger

Lists all the functions, filters and variables of the app.

```
$ php bin/console debug:twig
```

```
Functions
```

```
...
```

```
Filters
```

```
...
```

```
Tests
```

```
...
```

```
Globals
```

```
...
```

# Error pages

# Error pages in Symfony

- Symfony treats all errors as **exceptions** (e.g. a 404 error is a `NotFoundHttpException`)
- Error pages are rendered by a **ExceptionController** included in the **TwigBundle**.

# How is the error template selected?

- **error + status code + format + twig**  
(error404.json.twig, error500.xml.twig)
- **error + format + twig**  
(error.json.twig, error.xml.twig)
- **error.html.twig**

The first template that exists is used.

# Use your own error pages

- You must override the default templates used by TwigBundle.
- Bundle templates are overridden in:  
templates/bundles/<BundleName>/<template\_path>

Example:

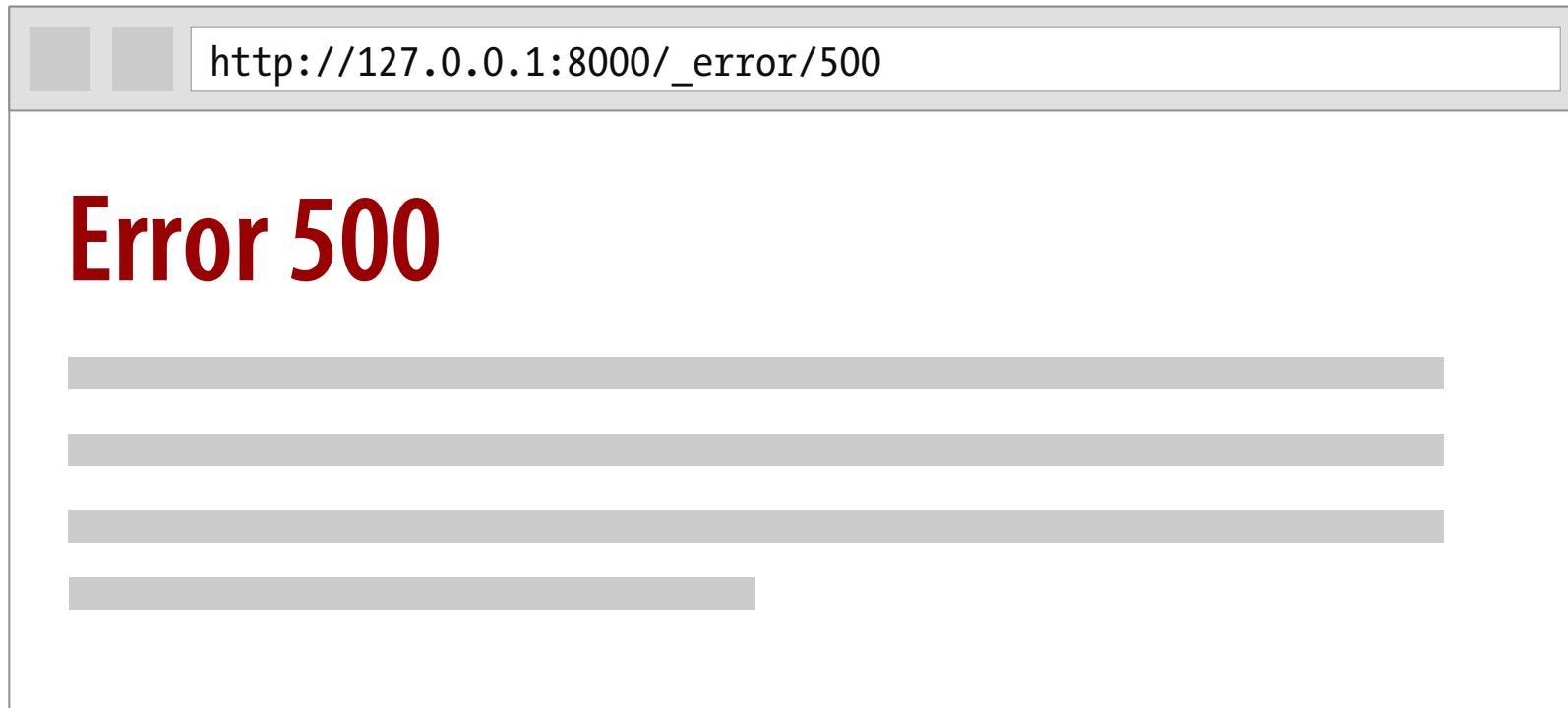
templates/bundles/TwigBundle/Exception/error404.html.twig

# Overriding error templates

```
templates/
└ bundles/
    └ TwigBundle/
        └ Exception/
            ├── error404.html.twig
            ├── error403.html.twig
            ├── error.html.twig
            ├── error404.json.twig
            ├── error403.json.twig
            └── error.json.twig
```

# Preview error pages

In the **dev** environment, browse  
**/\_error/{status\_code}**





# i18n Internationalization

# Basic concepts

# Internationalization

The process of **abstracting strings** and other locale-specific pieces out of your application into a layer where they can be **translated** and **converted** based on the user's locale.

# Locale

**Locale = Language + Country**

- ISO 639-1 defines language codes  
[https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639-1\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)
- ISO-3166-1 alpha-2 defines country codes  
[https://en.wikipedia.org/wiki/ISO\\_3166-1](https://en.wikipedia.org/wiki/ISO_3166-1)

# Locale examples

	Language	Country
en_AU	English	Australia
en_GB	English	United Kingdom
en_US	English	United States

It's common for the **locale** to only define the first language part (**en**, **fr**, etc.)

	Language	Country
fr_FR	French	France
fr_BE	French	Belgium

# Internationalization workflow

# Workflow

1. Enable and configure translation.
2. Extract content strings.
3. Create/update translation files.
4. Manage user locale.

Steps 1 and 4 are one-time tasks. Steps 2 and 3 are repeated continuously as long as the application grows and evolves.

Step 1.  
Enable translation  
and configure it

# Install the translation component

```
$ cd my-project/  
$ composer require translation
```

# Configure the translation component (1/2)

```
# config/packages/translation.yaml
framework:
    default_locale: '%locale%'
    translator:
        default_path: '%kernel.project_dir%/translations'
    fallbacks:
        - '%locale%'
```

The **default locale** is used when no locale is explicitly defined by the user (you can only define one).

# Configure the translation component (2/2)

```
# config/packages/translation.yaml
framework:
    default_locale: '%locale%'
    translator:
        default_path: '%kernel.project_dir%/translations'
        fallbacks:
            - '%locale%'
```

If a content is not available in the current locale, it is translated into the **fallback locales** (you can define more than one).

Step 2.  
Extract  
content strings

# Translating contents in controllers

```
use Symfony\Component\Translation\TranslatorInterface;  
  
public function index(TranslatorInterface $translator)  
{  
    $title = $translator->trans('Contact us');  
}
```

If the user's locale is `fr_FR` and there is a catalogue of french translations, `$title` value will be **Contactez-nous**.

In normal services (not controllers) inject the same `TranslatorInterface` service in the service class **constructor**.

# Translating template contents

```
{% trans %}
```

Contact us

```
{% endtrans %}
```

```
{{ 'Contact us' | trans }}
```

Use Twig tags to translate large blocks of static contents.

Use Twig filters to translate variables and expressions.

# Main difference between filters and tags

```
{% trans %}  
<h1>Contact us</h1>  
{% endtrans %}
```

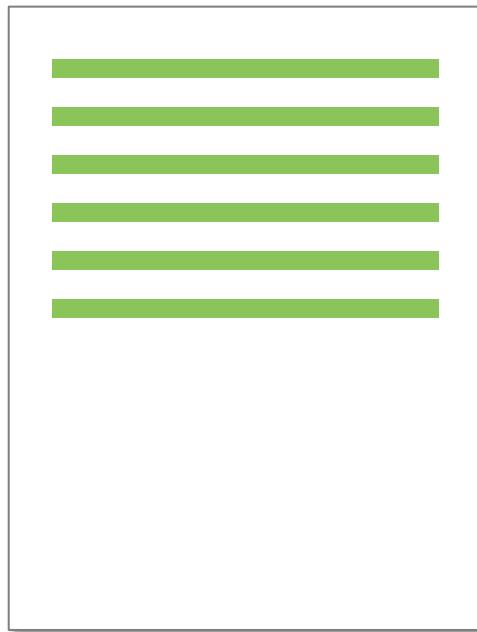
**OUTPUT** <h1>Contactez-nous</h1>

```
{{ '<h1>Contact us</h1>' |trans }}
```

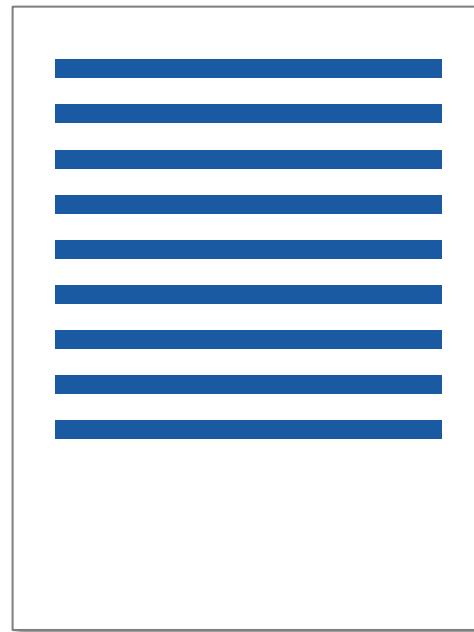
**OUTPUT** &lt;h1&gt;Contactez-nous&lt;/h1&gt;

# Step 3. Create translation files

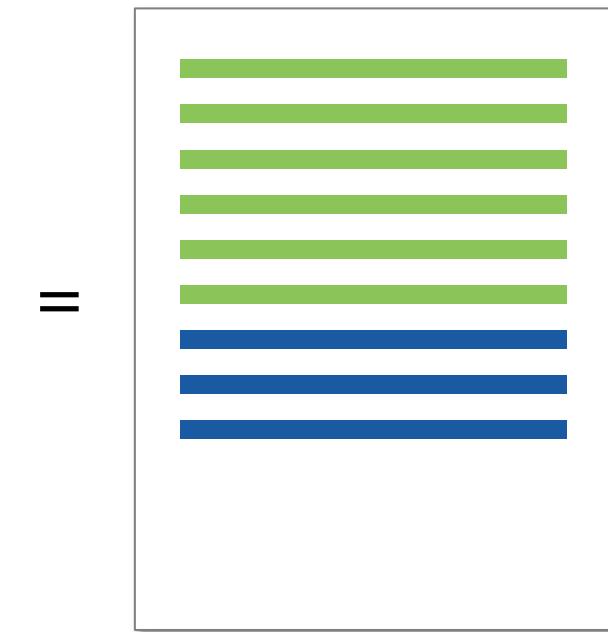
# How does Symfony get the translation



User's locale  
translations



Fallback locale  
translations



Complete translation file  
used by Symfony

# Translation files naming syntax

messages.fr\_FR.xlf



Domain  
(explained later)

Locale  
(it's common to define just the language without the country)

File format  
(XLIFF, YAML, PHP, PO/MO, etc.)

# Translation files location

```
your-project/
```

```
  └── ...
  └── src/
  └── templates/
  └── tests/
  └── translations/
      ├── messages.fr.xlf
      └── validators.fr.xlf
  └── var/
  └── ...
```

The **translations/** dir has highest priority and it **overrides** any other catalogue with the same name and locale, regardless of where it's defined originally. It is possible to override any third-party bundle translation catalogue.

# The XLIFF translation format

- Symfony Best Practices recommend to use this format.
- Pro: it's the standard format in the translation industry.
- Con: it's very verbose (it's based on XML)

# An example of XLIFF translation file

```
<!-- translations/messages.fr_FR.xlf -->
<?xml version="1.0" encoding="utf-8"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:1.2" version="1.2">
<file source-language="en" target-language="fr" datatype="plaintext" original="file.ext">
    <body>
        <trans-unit id="1">
            <source>Login</source>
            <target>Identifiez-vous</target>
        </trans-unit>
        <trans-unit id="2">
            <source>Username</source>
            <target>Nom d'utilisateur</target>
        </trans-unit>
        <trans-unit id="3">
            <source>Password</source>
            <target>Mot de passe</target>
        </trans-unit>
    </body>
</file>
</xliff>
```

# The YAML translation format

- Lots of Symfony developers use it.
- **Pro:** it's easy to read/write and supports nested messages.
- **Con:** it's not standard and its syntax is very strict (spaces vs. tabs, etc.)

# An example of YAML translation file

```
# translations/messages.fr_FR.yml  
Login: 'Identifiez-vous'  
Username: 'Nom d'utilisateur'  
Password: 'Mot de passe'
```

# Symfony supports lots of translation formats

- PHP Arrays
- CSV
- ICU (Data & RES)
- INI
- MO / PO
- Plain PHP
- QT
- XLIFF
- JSON
- YAML

# Translation strings vs Translation keys

```
<!-- messages.en.xlf -->
<trans-unit id="1">
    <source>An authentication exception occurred.</source>
    <target>An authentication exception occurred.</target>
</trans-unit>

<!-- messages.fr.xlf -->
<trans-unit id="2">
    <source>An authentication exception occurred.</source>
    <target>Une exception d'authentification s'est produite.</target>
</trans-unit>
```

**Translation strings** make catalogues easier to read, but any change in the original contents forces you to update the catalogues for all locales.

# Translation strings vs Translation keys

```
<!-- messages.en.xlf -->
<trans-unit id="1">
    <source>error.auth_exception</source>
    <target>An authentication exception occurred.</target>
</trans-unit>
```

```
<!-- messages.fr.xlf -->
<trans-unit id="2">
    <source>error.auth_exception</source>
    <target>Une exception d'authentification s'est produite.</target>
</trans-unit>
```

Symfony's Best Practices recommend to use keys.

**Translation keys** simplify translation management because you can change the original contents without updating the rest of catalogues.

# Step 4. Manage user locale

# Getting the user's locale

```
use Symfony\Component\HttpFoundation\Request;  
  
public function index(Request $request)  
{  
    $locale = $request->getLocale();  
}
```

The locale is stored in the **Request**, which means that it's not "sticky" and you must get its value for every request.

# Setting the user's locale via the URL

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class DefaultController
{
    /**
     * @Route("/{_locale}/contact", name="contact")
     */
    public function contact(Request $request)
    {
        $locale = $request->getLocale();
        // ...
    }
}
```

**\_locale** (with a leading underscore) is a special routing parameter used by Symfony to set the user's locale.

# Setting the user's locale via the session

```
public function onKernelRequest(GetResponseEvent $event)
{
    $request = $event->getRequest();

    // some logic to determine the $locale ...

    $request->getSession()->set('_locale', $locale);
}
```

This solution requires the use of **events** and **listeners**, which is out of the scope of this workshop.

Full details: [https://symfony.com/doc/current/session/locale\\_sticky\\_session.html](https://symfony.com/doc/current/session/locale_sticky_session.html)

# Forcing the translation locale in the controller

```
use Symfony\Component\Translation\TranslatorInterface;  
  
public function index(TranslatorInterface $translator)  
{  
    $title = $translator->trans(  
        'Contact us', array(), 'messages', 'fr_FR'  
    );  
}
```

Avoid this technique as much as possible and rely on the other natural ways of setting and getting the user's locale.

# Forcing the translation locale in the template

```
{% trans 'Contact us' locale='fr_FR' %}
```

```
{% trans into 'fr_FR' %}
```

Contact us

```
{% endtrans %}
```

Avoid this technique as much as possible and rely on the other natural ways of setting and getting the user's locale.

# Translating variable contents

# Translating messages that include variables

```
$message = "Hello $name";
```

Messages which contain **the value of some variable** are very common in web applications. How can you translate them?

# Translating variable messages in controllers

```
use Symfony\Component\Translation\TranslatorInterface;  
  
public function index(TranslatorInterface $translator)  
{  
    $title = $translator->trans(  
        'Hello %name%', ['%name%' => 'John']  
    );  
}
```

Variable parts are called **placeholders**. The wrapping % ... % characters are optional but used by convention.

# Translating variable messages in templates

```
{{ 'Hello %name%' | trans({  
    '%name%': 'John'  
}) }}
```

```
{% trans with {'%name%': 'John'} %}
```

```
    Hello %name%  
{% endtrans %}
```

Variable parts are called **placeholders**. The wrapping % ... % characters are optional but used by convention.

# Translating XLIFF messages with variable parts

```
<!-- translations/messages.fr_FR.xlf -->
<?xml version="1.0"?>
<xliff version="1.2"
      xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" target-language="fr"
        datatype="plaintext" original="file.ext">
    <body>
      <trans-unit id="1">
        <source>Hello %name%</source>
        <target>Bonjour %name%</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

# Translating YAML messages with variable parts

```
# translations/messages.fr_FR.yml
'Hello %name%': 'Bonjour %name%'
```

# Translations based on variables

# Translating plural messages

```
$singular = 'There is one product left.';  
$plural = 'There are %count% products left.';
```

Most languages have simple **pluralization rules**, but some of them (e.g. Russian) define very complex rules.

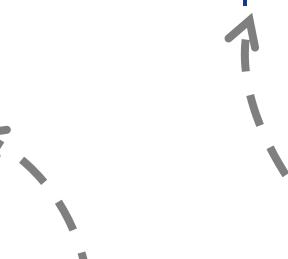
Symfony abstracts this issue and provides out-of-the-box pluralization support for most of the world's languages.

# Translating plural messages in controllers

```
use Symfony\Component\Translation\TranslatorInterface;
```

```
public function index(TranslatorInterface $translator)
{
    $title = $translator->trans(
        'There is one product left.|There are %count%
products left.',
        ['%count%' => 10]
    );
}
```

Value of `%count%` is considered to decide which message to pick (singular or plural).



Message alternatives are separated with a pipe (|)

# Translating plural messages in templates

```
{% trans with {'%count%': 10} %}
```

There is one product left.|There are %count% products left.

```
{% endtrans %}
```

```
{{ 'There is one product left.|There are %count% products left.' |trans({'%count%':10}) }}
```

# Understanding the pluralization rules

// English

'There is one product left.

→ |There are %count% products left.'

If **count = 0**, Symfony selects ...

// French

'Il y a %count% produit.

|Il y a %count% produits.'

# Understanding the pluralization rules

// English

→ 'There is one product left.  
| There are %count% products left.'

If **count = 1**, Symfony selects ...

// French

→ 'Il y a %count% produit.  
| Il y a %count% produits.'

# Understanding the pluralization rules

// English

'There is one product left.

→ |There are %count% products left.'

If **count > 1**, Symfony selects ...

// French

'Il y a %count% produit.

→ |Il y a %count% produits.'

# Explicit interval pluralization

// English

```
'{0} There is no product left.|{1} There is  
one product left.|[1,Inf] There are %count%  
products left.'
```

// French

```
'{0, 1} Il y a %count% produit.| ]1,Inf] Il y  
a %count% produits.'
```

It's **optional**, but most of the times it helps to better understand which message will be selected.

# Explicit interval pluralization

]-Inf, 0] C'est fini, vous n'avez plus d'essai !

|{1} Attention, c'est votre dernière chance !

|[2,5] Méfiez-vous, il vous reste %count% essais restants !

|[6,8] Pas de panique, vous avez encore %count% essais restants !

|[9, +Inf[ Vous avez encore %count% essais restants !

Intervals are defined using the ISO 31-11 standard.

Full Details: [en.wikipedia.org/wiki/Interval\\_\(mathematics\)#Notations\\_for\\_intervals](https://en.wikipedia.org/wiki/Interval_(mathematics)#Notations_for_intervals)

# Full reference of trans()

```
$translator->trans(  
    'There is one product left.|There are %count% products left.',  
    ['%count%' => 10],  
    'admin',  
    'fr_FR'  
) ;
```

# Full reference of |trans and { % trans % }

```
{ { message|trans } }
```

```
{ { message|trans({ '%count%' : 10}, 'admin', 'fr_FR') } }
```

```
{% trans with {'%count%':count} from 'admin' into 'fr_FR' %}
```

'There is one product left.|There are %count% products left.'

```
{% endtrans %}
```

Creating / updating  
translation files  
automatically

# Log missing translations

```
# config/packages/translation.yaml
framework:
    translator:
        logging: true

# var/log/dev.log
[201X-04-20 15:06:43] translation.WARNING: Translation not found.
{"id": "Title", "domain": "messages", "locale": "en"}
[201X-04-20 15:06:43] translation.WARNING: Translation not found.
{"id": "Summary", "domain": "messages", "locale": "en"}
[201X-04-20 15:06:43] translation.WARNING: Translation not found.
{"id": "Content", "domain": "messages", "locale": "en"}
[201X-04-20 15:06:43] translation.WARNING: Translation not found.
{"id": "Author email", "domain": "messages", "locale": "en"}
```

# Show unused or missing translations

```
$ php bin/console debug:translation fr
```

State	Id	Message Preview (fr)
	title.post_list	Liste des articles
missing	action.show	action.show
fallback	action.edit	Edit
unused	action.create_post	Créer un nouvel article

# Create the translation catalogues

```
$ php bin/console translation:update en --dump-messages
```

```
Generating "en" translation files for "default directory"
```

```
Parsing templates
```

```
Loading translation files
```

```
Displaying messages for domain messages:
```

```
title.post_list
```

```
action.show
```

```
action.edit
```

```
action.create_post
```

# Create the translation catalogues

```
$ php bin/console translation:update en --force
```

```
Generating "en" translation files for "default directory"
```

```
Parsing templates
```

```
Loading translation files
```

```
Writing files
```

```
<!-- translations/messages.en.xlf -->
<trans-unit id="04a6524e12dc0bad0a3146c8" resname="title.post_list">
    <source>title.post_list</source>
    <target>_title.post_list</target>
</trans-unit>
```

# Create the translation catalogues

```
$ php bin/console translation:update en --force --prefix=new_
```

```
Generating "en" translation files for "default directory"
```

```
Parsing templates
```

```
Loading translation files
```

```
Writing files
```

```
<!-- translations/messages.en.xlf -->
<trans-unit id="04a6524e12dc0bad0a3146c8" resname="title.post_list">
    <source>title.post_list</source>
    <target>new_title.post_list</target>
</trans-unit>
```

# Create the translation catalogues

```
$ php bin/console translation:update en --force --prefix=new_  
--output-format=yaml
```

Generating "en" translation files for "default directory"

Parsing templates

Loading translation files

Writing files

```
# translations/messages.en.yaml  
title.post_list: new_title.post_list  
action.show: new_action.show  
action.edit: new_action.edit  
action.create_post: new_action.create_post
```

# Create the translation catalogues

```
$ php bin/console translation:update en --force --prefix=new_  
--output-format=yaml AppBundle
```

Generating "en" translation files for "AppBundle"

Parsing templates

Loading translation files

Writing files

```
# translations/messages.en.yaml  
title.post_list: new_title.post_list  
action.show: new_action.show  
action.edit: new_action.edit  
action.create_post: new_action.create_post
```

# Translation domains

# Translation domains

- An optional way to organize messages into groups.
- By default, all messages are grouped in a domain called "messages".
- In most applications there is no need or justification for using several domains.

# Selecting the domain in the controller

```
$translator->trans(  
    'Contact us',  
    array(),  
    'admin'  
) ;
```

The translation is stored in the  
`admin.fr_FR.<format>` file

If different from "messages", set the translation domain as the third optional argument of the `trans()` method.

# Selecting the domain in the template

```
{ { 'Contact us' | trans(domain='admin') } }
```

```
{% trans from 'admin' %}
```

Contact us

```
{% endtrans %}
```

The translation is stored in the `admin.fr_FR.<format>` file.

# Selecting the default domain in the template

```
{% trans_default_domain 'admin' %}
```

```
{# ... template contents ... #}
```

Note that this only influences **the current template**, not any "included" template (in order to avoid side effects).

# Form and database translation

# Translating form validation messages

```
// src/Entity/User.php
use Symfony\Component\Validator\Constraints as Assert;

class User {
    /**
     * @Assert\NotBlank(message = "user.name.not_blank")
     */
    public $name;
}
```

```
<!-- validators.en.xlf -->
<trans-unit id="1">
    <source>user.name.not_blank</source>
    <target>Please enter the name of the user.</target>
</trans-unit>
```

# Translating database contents

- This feature is not provided by the translation component.
- Install **StofDoctrineExtensionsBundle**  
<https://github.com/stof/StofDoctrineExtensionsBundle>
- Use **Translatable** extension.



# Introduction to Forms

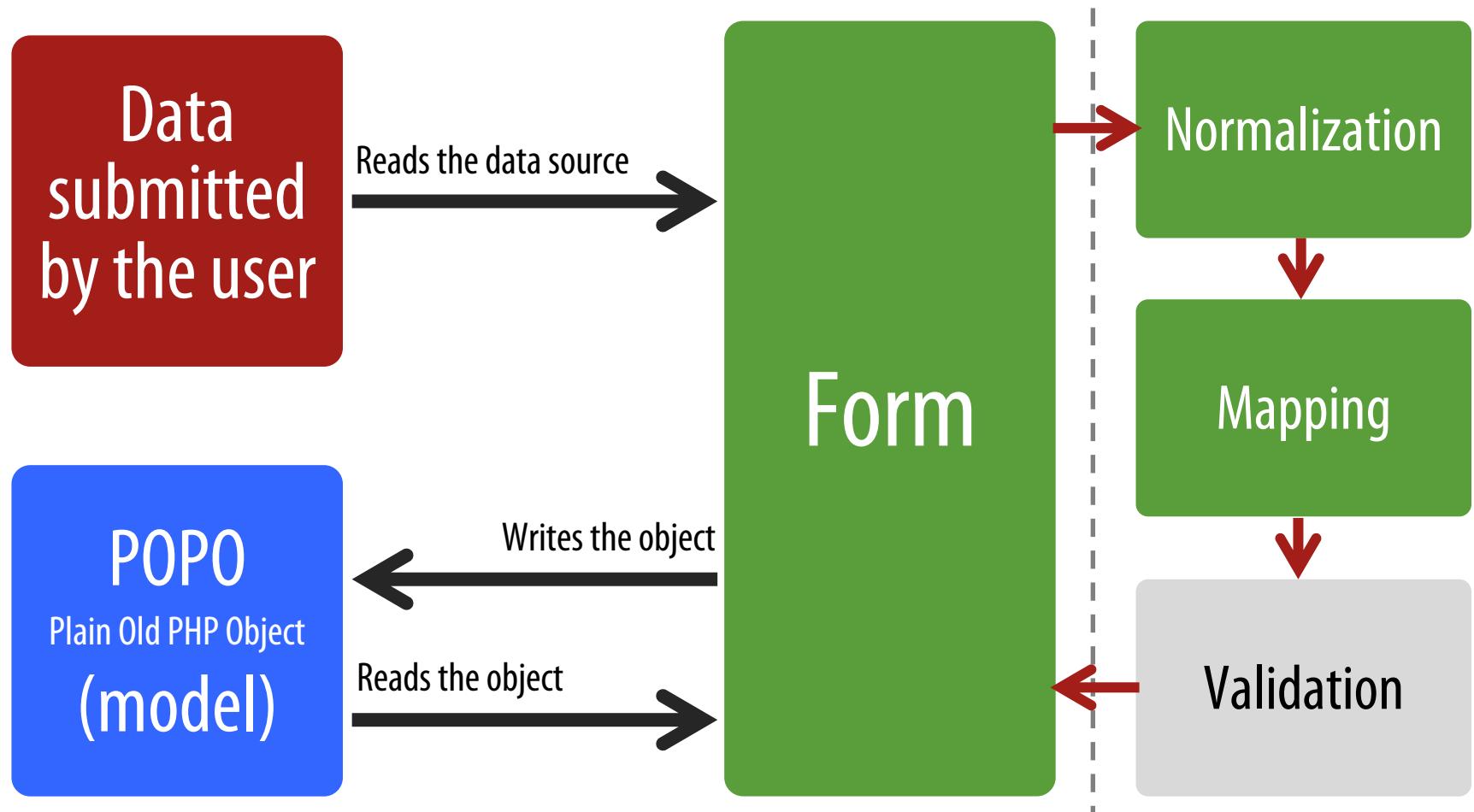
# Adding form support in Symfony projects

# Adding form support in Symfony projects

```
$ cd my-project/  
$ composer require form
```

# Basic concepts

# Symfony Form Component workflow



# Symfony Form Component architecture

**View**

Twig

PHP Templating

**Extensions**

Core

Validation

DI

CSRF

Doctrine

**Foundation**

Form Component

PropertyAccess Component

OptionsResolver Component

EventDispatcher Component

Source: <http://webmozarts.com/2012/03/06/symfony2-form-architecture/>

# The domain object

# Creating the domain object class

```
namespace App\Entity;  
  
class Product  
{  
    public $name;  
    private $price;  
  
    public function setPrice($price)  
    {  
        $this->price = (float) $price;  
    }  
  
    public function getPrice()  
    {  
        return $this->price;  
    }  
}
```

Symfony forms manipulate the information stored in **plain PHP objects (POPO)**.

The only requirement is that **properties must be public** or define a **getter/isser + setter**.

# Building the form

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product)
            ->add('name', TextType::class)
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm();

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

## CAUTION

It's not recommended to build forms in the controller unless they are trivial and used once.

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product)
            ->add('name', TextType::class)
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm();

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

1. Create or look for the object that is edited with the form. The properties of the object initialize the form fields.

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        1 $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product) 2
            ->add('name', TextType::class)
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm();

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

2. Use the `createFormBuilder()` shortcut to build the form object interactively by chaining `add()` method calls.

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        1 $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product) 2
            ->add('name', TextType::class)
        3 ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm();

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

3. Use the `add()` method to configure the form fields and their properties.

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product) ②
            ->add('name', TextType::class)
        ③ ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm(); ④

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

4. Invoke the `getForm()` after adding all form fields to create the actual Form object.

# Building the form in the controller

```
class ProductController extends AbstractController
{
    public function new()
    {
        $product = new Product();
        1 $product->name = 'Test product';
        $product->setPrice(50.00);

        $form = $this->createFormBuilder($product) 2
            ->add('name', TextType::class)
        3 ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
            ->getForm(); 4

        return $this->render('product/new.html.twig', [
            'form' => $form->createView(), 5
        ]);
    }
}
```

5. Templates cannot display Form objects directly. Use the `createView()` method to get the form's visual representation.

# Building the form in a separate class

```
// src/Form/ProductType.php
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
    }
}
```

# Building the form in a separate class

```
// src/Form/ProductType.php
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ProductType extends AbstractType ①
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
    }
}
```

1. Custom types must extend from **AbstractType** and implement **buildForm()**.

# Building the form in a separate class

```
// src/Form/ProductType.php
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('price', MoneyType::class, ['currency' => 'USD'])
            ->add('send', SubmitType::class, ['label' => 'Create the product'])
        ;
    }
}
```

2. Use the **\$builder** object to build the form chaining all the **add()** methods. There is no need to invoke **getForm()** at the end.

# Using a form class in the controller

```
use App\Entity\Product;
use App\Form\ProductType;

public function new()
{
    $product = new Product();
    $product->name = 'A name';
    $product->setPrice(50.00);

    $form = $this->createForm(ProductType::class, $product);

    // ...
}
```

# Using a form class in the controller

```
use App\Entity\Product;
use App\Form\ProductType;

public function new()
{
    ① $product = new Product();
    $product->name = 'A name';
    $product->setPrice(50.00);

    $form = $this->createForm(ProductType::class, $product);

    // ...
}
```

1. Create or look for the object that is edited with the form. The properties of the object initialize the form fields.

# Using a form class in the controller

```
use App\Entity\Product;
use App\Form\ProductType;

public function new()
{
    ① $product = new Product();
    $product->name = 'A name';
    $product->setPrice(50.00);

    ② $form = $this->createForm(ProductType::class, $product);

    // ...
}
```

2. Create the actual Form object with the `createForm()` shortcut. The first argument is the form type and the second argument is the object manipulated with the form.

# Built-in Symfony Form Types

## Text Fields

- ColorType
- EmailType
- IntegerType
- MoneyType
- NumberType
- PasswordType
- PercentType
- RangeType
- SearchType
- TelType
- TextareaType
- TextType
- UrlType

## Choice Fields

- ChoiceType
- EntityType
- CountryType
- LanguageType
- LocaleType
- TimezoneType
- CurrencyType

## Date and Time Fields

- DateType
- DateIntervalType
- DateTimeType
- TimeType
- BirthdayType

## Other Fields

- CheckboxType
- FileType
- RadioType

## Field Groups

- CollectionType
- RepeatedType

## Hidden Fields

- HiddenType

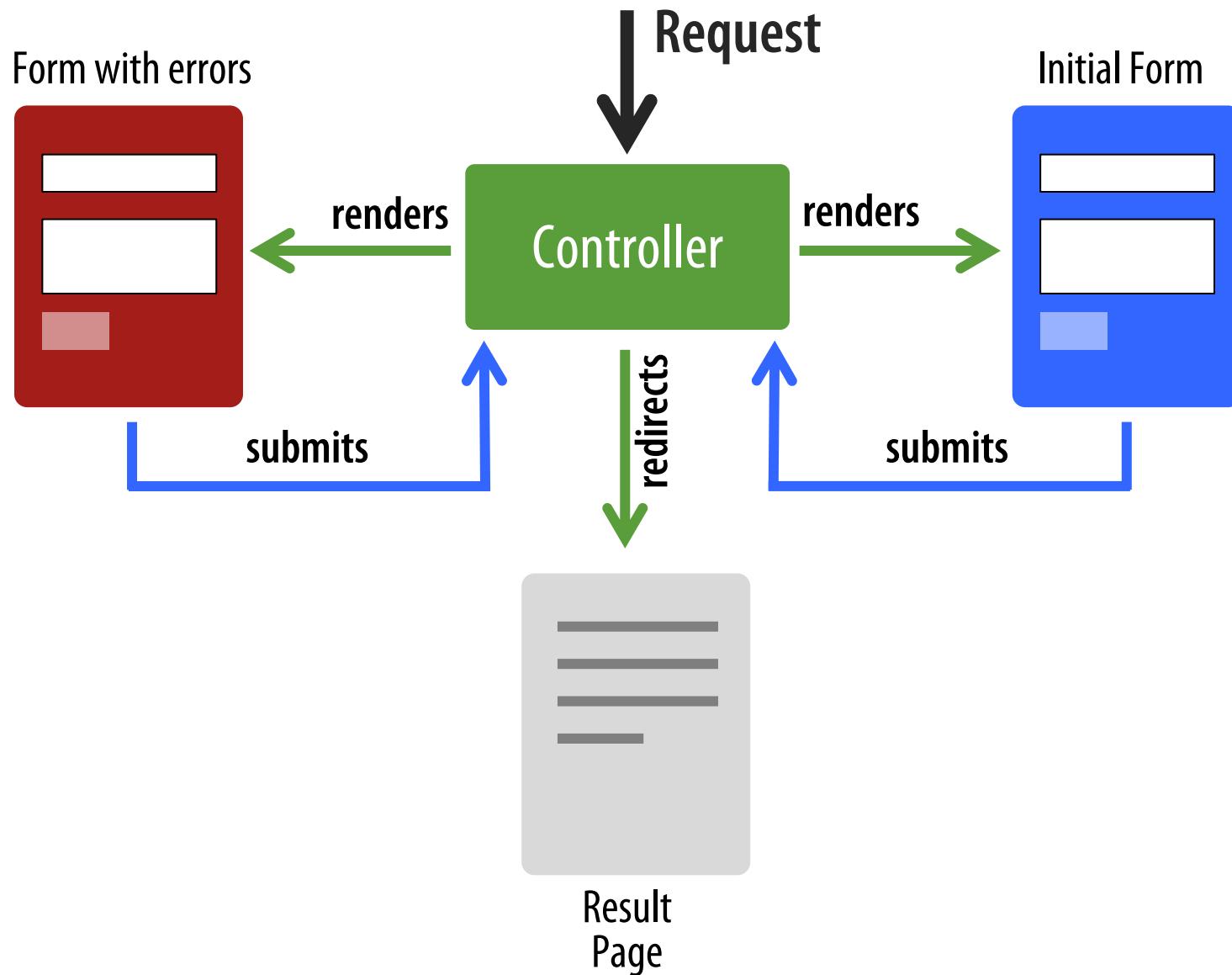
## Buttons

- ButtonType
- ResetType
- SubmitType

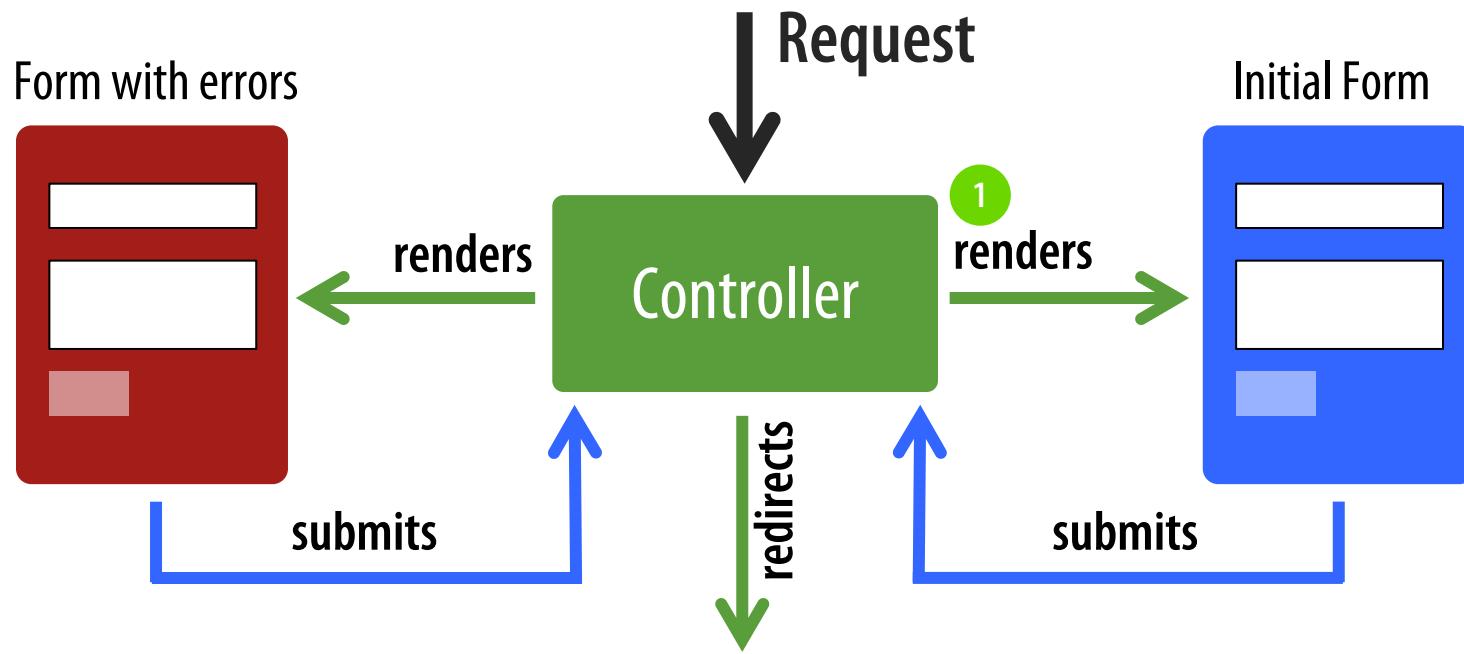
Full details: [symfony.com/doc/current/reference/forms/types.html](https://symfony.com/doc/current/reference/forms/types.html)

# Processing forms

# The Big Picture of handling Symfony forms



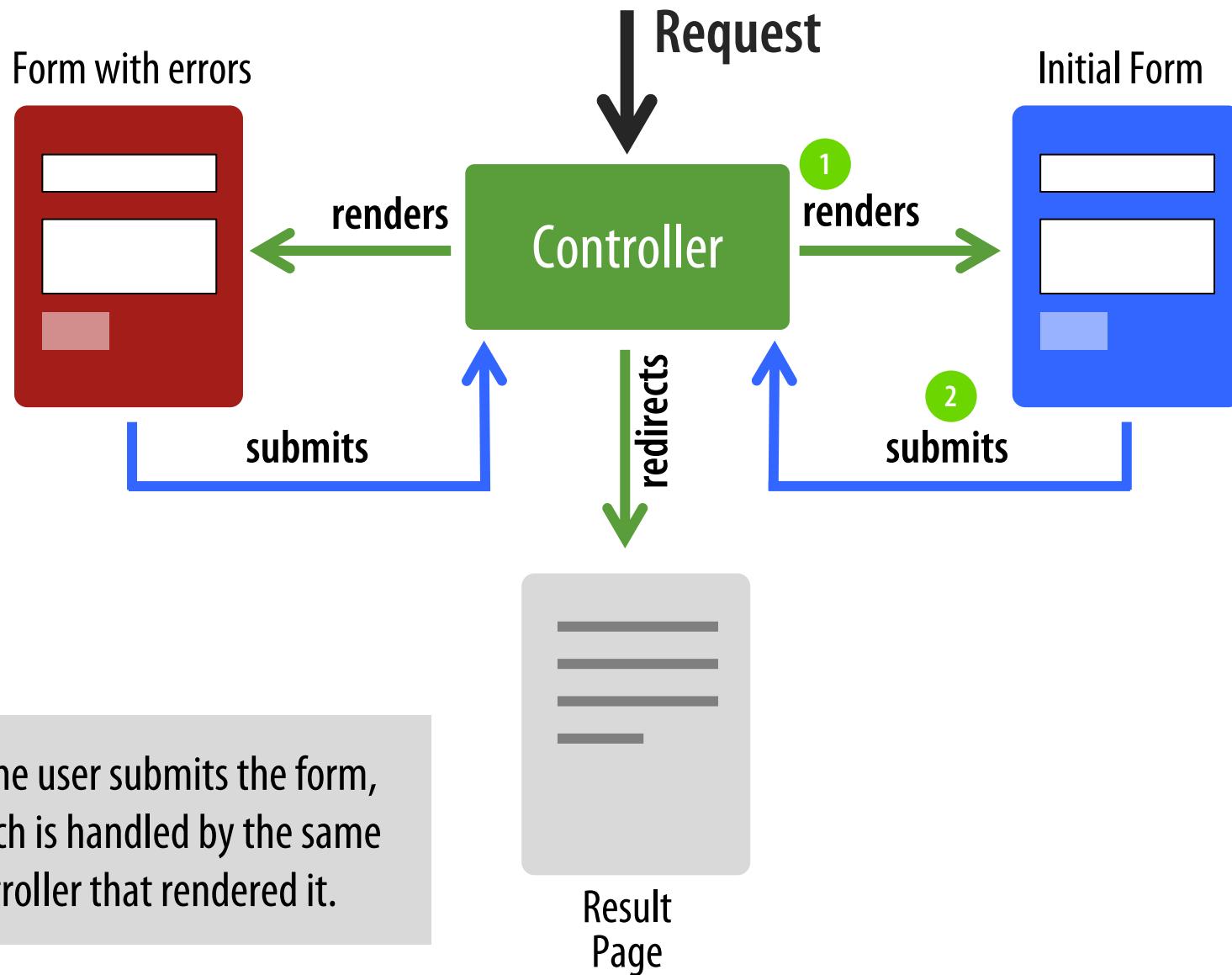
# The Big Picture of handling Symfony forms



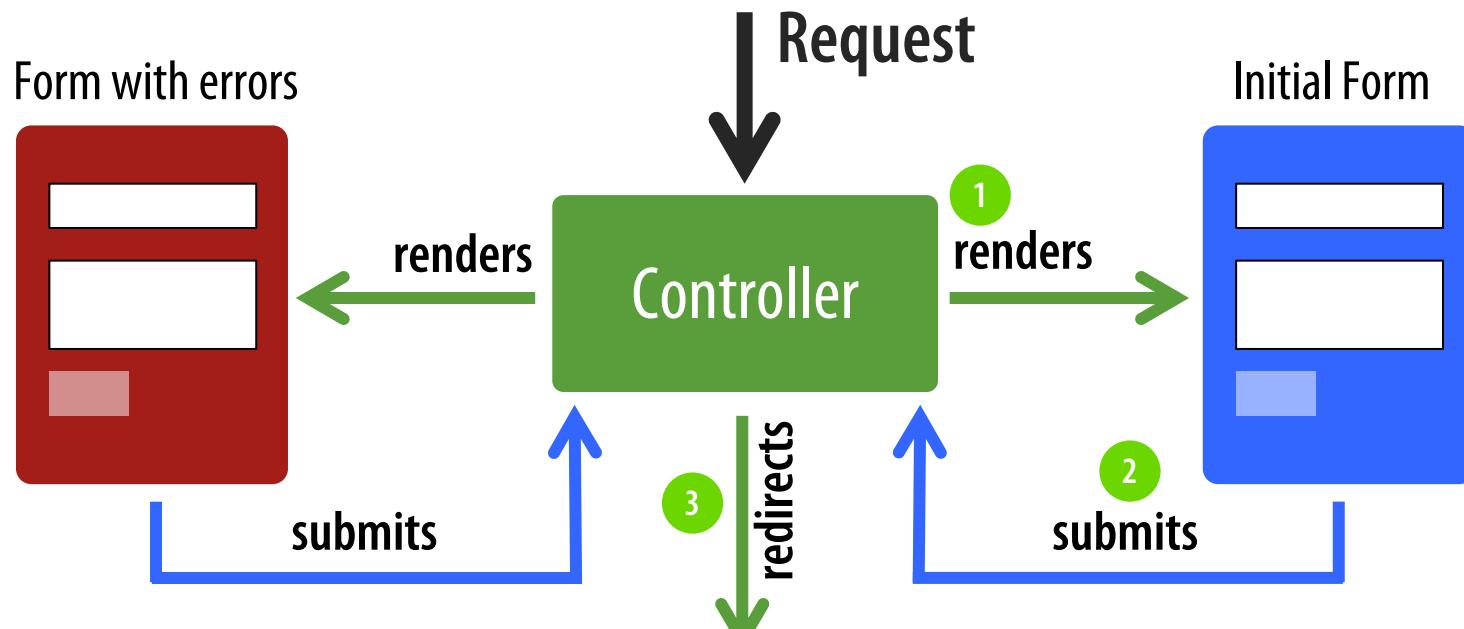
1. The controller serves the request rendering the initial form (it can be empty or prepopulated depending on the object passed to the form)

Result  
Page

# The Big Picture of handling Symfony forms



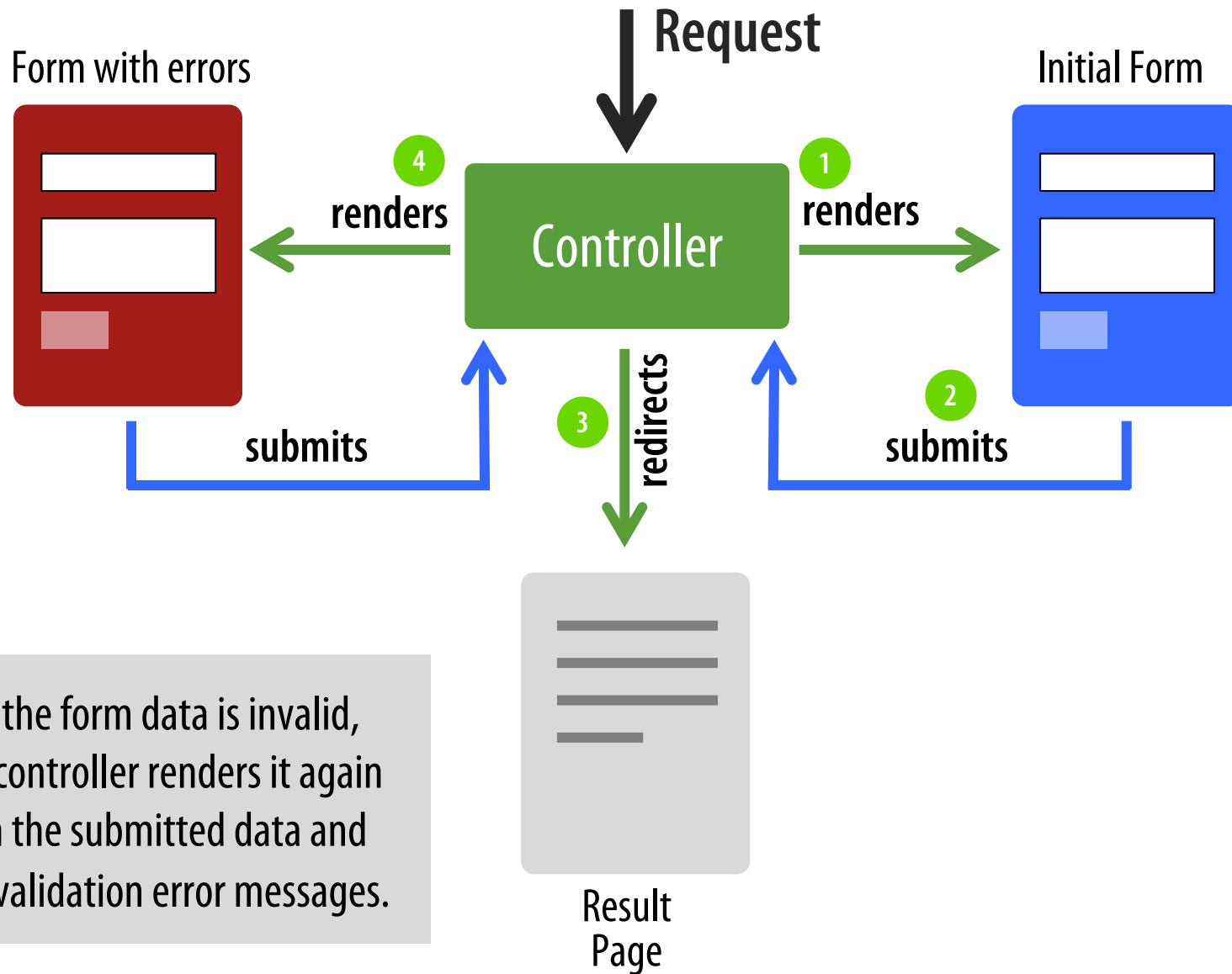
# The Big Picture of handling Symfony forms



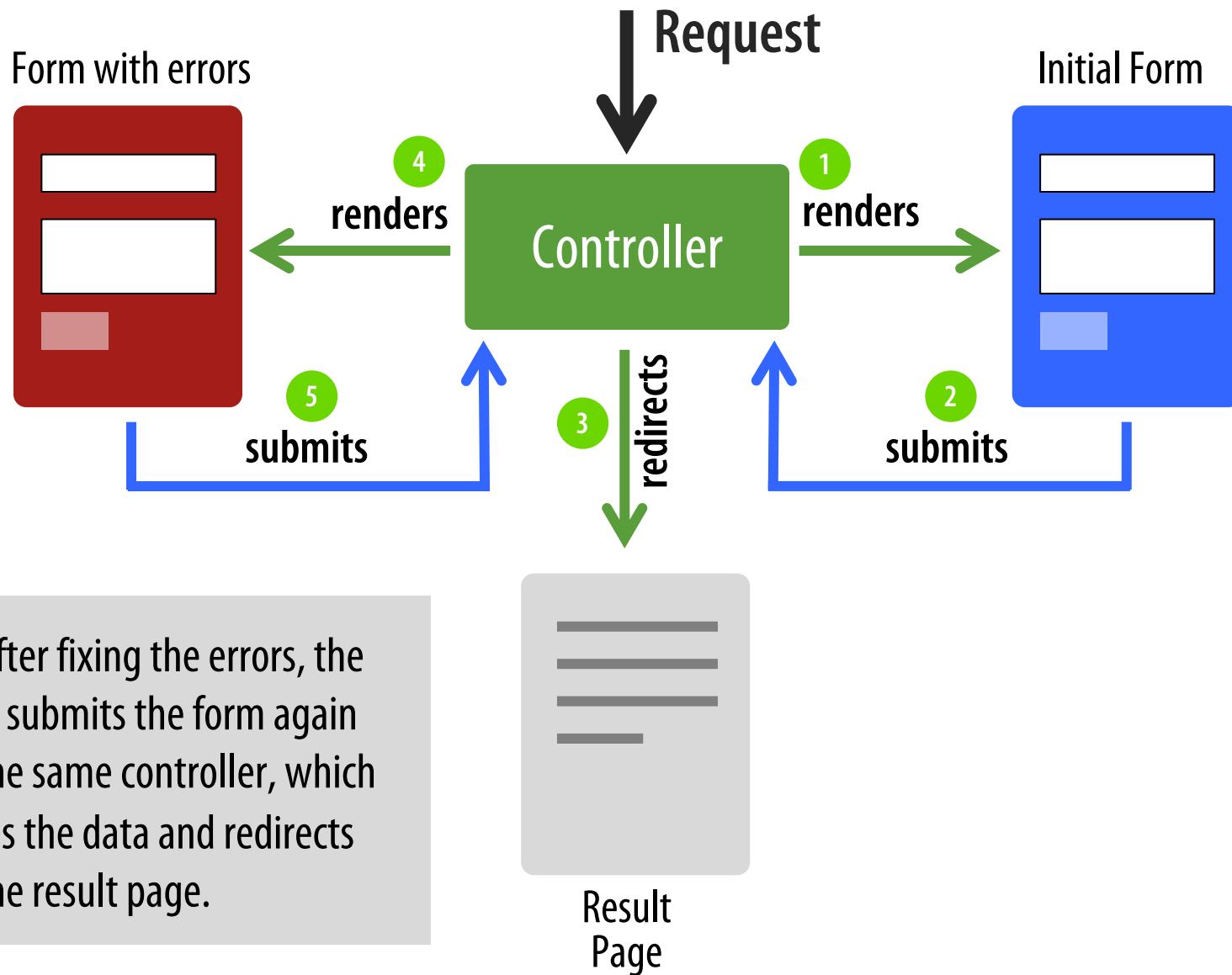
3. If the form data is valid, the controller saves the data and redirects the user to the result page (to avoid submitting the form again if page reloads).

Result  
Page

# The Big Picture of handling Symfony forms



# The Big Picture of handling Symfony forms



# Handling and processing forms in practice

```
use App\Entity\Product;
use App\Form\ProductType;

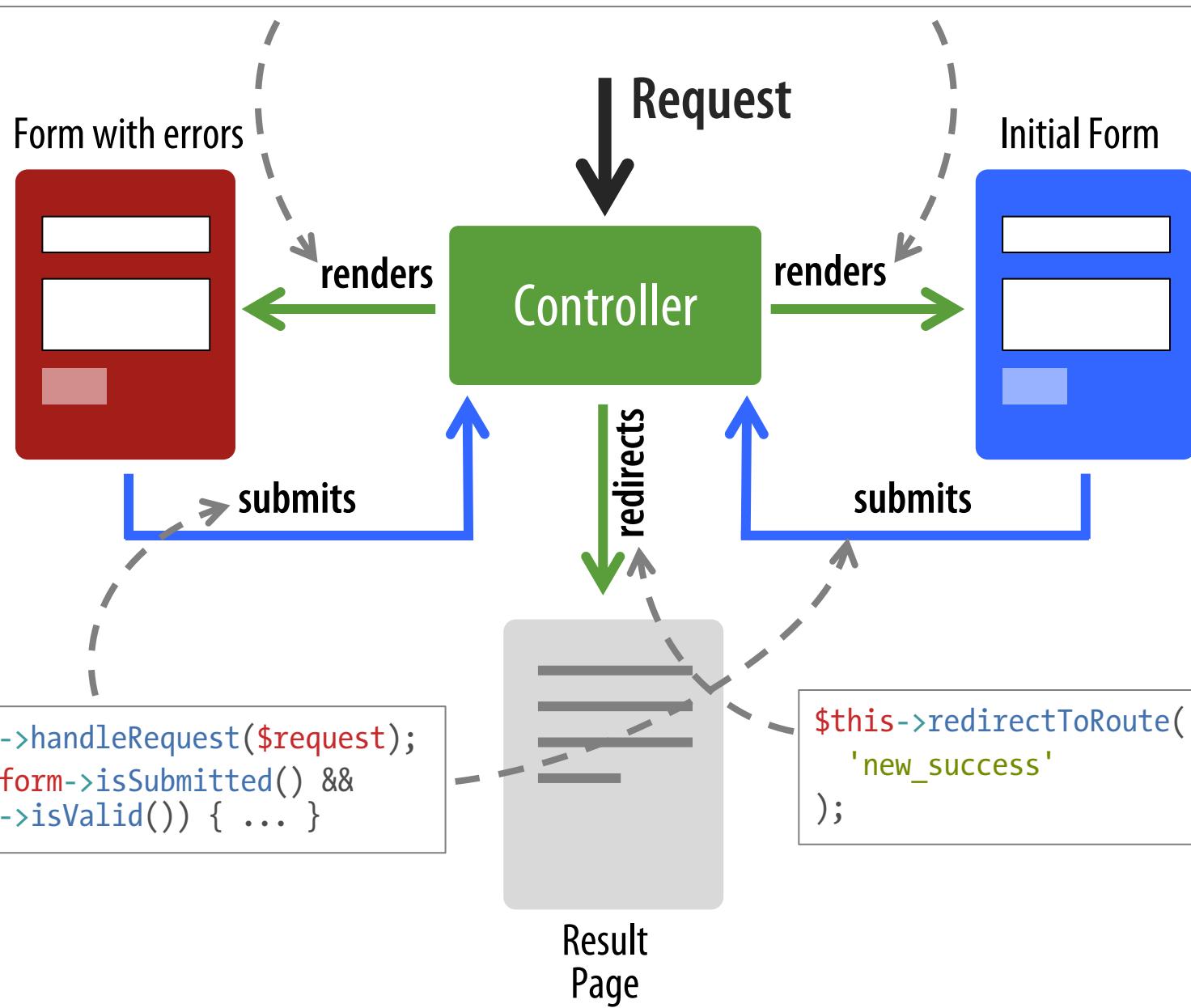
public function new(Request $request)
{
    $product = new Product();
    $form = $this->createForm(ProductType::class, $product);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // handle data, persist the object to the database...

        return $this->redirectToRoute('new_success');
    }

    return $this->render('product/new.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

```
$this->render('product/new.html.twig', ['form' => $form->createView()]);
```



# Rendering forms

# Fast form rendering for prototypes

```
 {{ form(form_view) }}
```

The **form()** function is a Twig extension provided by Symfony. It renders the labels, widgets and error messages for all form fields.

It's the fastest and easiest way to render a form, but it doesn't provide fine-grained control to tweak how the form is displayed.

# Advanced form rendering

```
 {{ form_start(form_view) }}
```

```
 {{ form_errors(form_view) }}
```

```
 {{ form_row(form_view.name) }}
```

```
 {{ form_row(form_view.price) }}
```

```
 {{ form_end(form_view) }}
```

# Advanced form rendering

```
{{ form_start(form_view) }}
```

It renders the `<form>` starting tag, sets the **action** and **method** attributes and adds, if necessary, the **enctype** attribute.

```
{{ form_end(form_view) }}
```

It renders the `</form>` ending tag and any form field which hasn't been explicitly rendered by the template. This is very useful to render hidden fields (e.g. CSRF token).

# Advanced form rendering

```
{{ form_errors(form_view) }}
```

It renders the global error messages associated with the form instead of a specific form field. You can "redirect" errors from fields to the form.

```
{{ form_row(form_view.name) }}
```

It renders the label, widget and error messages (if any) for the given form field.

# Configuring the form behavior

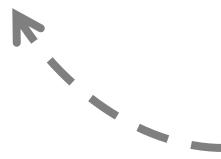
```
 {{ form(form_view, {  
     'action': '...',  
     'method': 'GET'  
}) }}
```

```
 {{ form_start(form_view, {  
     'action': '...',  
     'method': 'GET'  
}) }}
```

By default, these functions use the **POST method** and an empty **action attribute** to submit the form to the originating controller.

# Detailed form rendering

```
 {{ form_start(form_view) }}  
 {{ form_errors(form_view) }}  
  
<div>  
 {{ form_label(form_view.name) }}  
 {{ form_errors(form_view.name) }}  
 {{ form_widget(form_view.name) }}  
</div>  
 {{ form_end(form_view) }}
```



In this example, **form\_end()** displays the second form field.

# Detailed form rendering

```
{{ form_label(form_view.name) }}
```

It renders the label for the given form field.

```
{{ form_errors(form_view.name) }}
```

It renders the errors specific to the given form field (if any).

```
{{ form_widget(form_view.name) }}
```

It renders the HTML widget that represents the given form field.

# Adding validation constraints

# Validation constraints as annotations

```
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Product
{
    /**
     * @Assert\NotBlank()
     * @Assert\Length(max = 40)
     */
    public $name;

    /**
     * @Assert\NotBlank()
     * @Assert\Range(min = 1)
     */
    private $price;
}
```

Symfony Best Practices recommend to use **annotations** for validation, but YAML and XML are also supported.

# Validation constraints as a YAML file

```
# config/validator/validation.yaml
App\Entity\Product:
    properties:
        name:
            - NotBlank: ~
            - Length: { max: 40 }
        price:
            - NotBlank: ~
            - Range: { min: 1 }
```

# Validation constraints as an XML file

```
<!-- config/validator/validation.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mapping>
    <class name= "App\Model\Product">
        <property name="name">
            <constraint name="NotBlank"/>
            <constraint name="Length">
                <option name="max">
                    <value>40</value>
                </option>
            </constraint>
        </property>
        <property name="price">
            <constraint name="NotBlank"/>
            <constraint name="Range">
                <option name="min">
                    <value>1</value>
                </option>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

# Organizing YAML/XML validation files

config/

- └ validator/

- └ validation.yaml

config/

- └ validator/

- ├ Author.yaml

- ├ Category.yaml

- ├ Comment.yaml

- └ Post.yaml

# Built-in Symfony Validation Constraints

## Basic Constraints

- NotBlank
- Blank
- NotNull
- IsNull
- IsTrue
- IsFalse
- Type

## String Constraints

- Email
- Length
- Url
- Regex
- Ip

## Uuid

## Comparison Constraints

- EqualTo
- NotEqualTo
- IdenticalTo
- NotIdenticalTo
- LessThan
- LessThanOrEqual
- GreaterThan
- GreaterThanOrEqual
- Range
- DivisibleBy

## Date Constraints

- Date

## DateTime

## Time

## Collection Constraints

- Choice
- Collection
- Count
- UniqueEntity
- Language
- Locale
- Country

## File Constraints

- File
- Image

## Financial Constraints

- Bic
- CardScheme
- Currency
- Luhn
- Iban
- Isbn
- Issn

## Other Constraints

- Callback
- Expression
- All
- UserPassword
- Valid

Full details: [symfony.com/doc/current/reference/constraints.html](https://symfony.com/doc/current/reference/constraints.html)

# Translating validation messages (1 of 2)

```
// src/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank(message = "author.name.not_blank")
     */
    public $name;
}
```

It's recommended to use keys as the content of the original messages, to make translations easier to maintain.

# Translating validation messages (2 of 2)

```
<!-- translations/validators.en.xlf -->
<?xml version="1.0"?>
<xliff version="1.2"
xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext"
original="file.ext">
    <body>
      <trans-unit id="author.name.not_blank">
        <source>author.name.not_blank</source>
        <target>Please enter an author name.</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

# Form debugging

# Form errors in the web debug toolbar

Author email  
anna\_admin@symfony.com

Published at  
201X-03-07T11:30:43+01:00

[Create post](#) [Save and create new](#) [Back to the post list](#)

© 2016 - The Symfony Project  
MIT License

200 @ admin\_post\_new 90 ms 6.5 MB  1  21  anna\_admin  34 ms  1 in 0.65 ms  3.x.y X



# Form errors in the Symfony profiler

Symfony Profiler

search on symfony.com Search

http://symfony-demo.dev/app\_dev.php/en/admin/post/new

Method: POST HTTP Status: 200 IP: ::1 Profiled on: Mon, 07 Mar 201X 11:31:24 +0100 Token: 0a87c2

Last 10 Latest Search

Request / Response Performance Forms 1 Exception Logs 34 Events Routing Translation Security Twig Doctrine E-Mails Debug Configuration

## Forms

post [AppBundle\Form\PostType]

- title [text] 1
- summary [textarea]
- content [textarea]
- authorEmail [email]
- publishedAt [DateTimePickerType]
- saveAndCreateNew [submit]
- \_token [hidden]

### title [text]

#### Errors

Message	Origin	Cause
This value should not be blank.	title	Symfony\Component\Validator\ConstraintViolation Object(Symfony\Component\Form\Form).data.title = null

#### Default Data

Property	Value
Model Format	same as normalized format
Normalized Format	null
View Format	

#### Submitted Data

Property	Value
View Format	
Normalized Format	null
Model Format	same as normalized format



# SensioLabs services

# About us

We are the **creators of Symfony**. We know the framework and PHP inside out and we can help you.

## Contact us

### SensioLabs France

92-98, Boulevard Victor Hugo

92 115 Clichy Cedex

France

Tel: +33 (0)1 40 99 81 09

[contact@sensiolabs.com](mailto:contact@sensiolabs.com)

### SensioLabs Deutschland

Neusser Str. 27-29

50670 Köln

Germany

Tel: +49 221 66 99 27 50

[contact@sensiolabs.de](mailto:contact@sensiolabs.de)

We also provide worldwide **on-site services**. Contact us at: [sensiolabs.com/en/contact](http://sensiolabs.com/en/contact)

# Our products and services

SensioLabs  
Insight

Blackfire

Consulting

Training

# SensioLabs Insight

[insight.sensiolabs.com](http://insight.sensiolabs.com)

SensioLabsInsight

Dashboard Help What we analyze Pricing Blog Account

fabpot / symfony #3812 Triggered by the API 2 hours ago, duration: 8 minutes

Violations (329) Fixed Ignored

▼ Database queries should use parameter binding Critical Security ?

in src/Symfony/Component/HttpKernel/Profiler/PdoProfilerStorage.php, line 59

```
54.  
55.     list($criteria, $args) = $this->buildCriteria($ip, $url, $start, $end);  
56.  
57.     $criteria = $criteria ? 'WHERE ' . implode(' AND ', $criteria) : '';
```

If provided by the user, the value of `implode(' AND ', $criteria)` may allow an SQL injection attack. Avoid concatenating parameters to SQL query strings, and use parameter binding instead.

Time to fix: about 1 hour

Comment Open Issue Permalink

Last edited 4 years ago by Jan Schumann

3 months to get the Platinum Medal

Search

Severity

2 Critical  
49 Major  
262 Minor  
16 Info

Category

38 Architecture  
97 Bugrisk  
17 Codestyle  
171 Deadcode  
4 Performance  
2 Security

Developer

117 Nicolas Grekas  
48 Bernhard Schussek

► Global variable or function should never be used Major Architecture ?

► Logical operators should be avoided Major Bugrisk ?

It analyzes the quality of your code continuously.

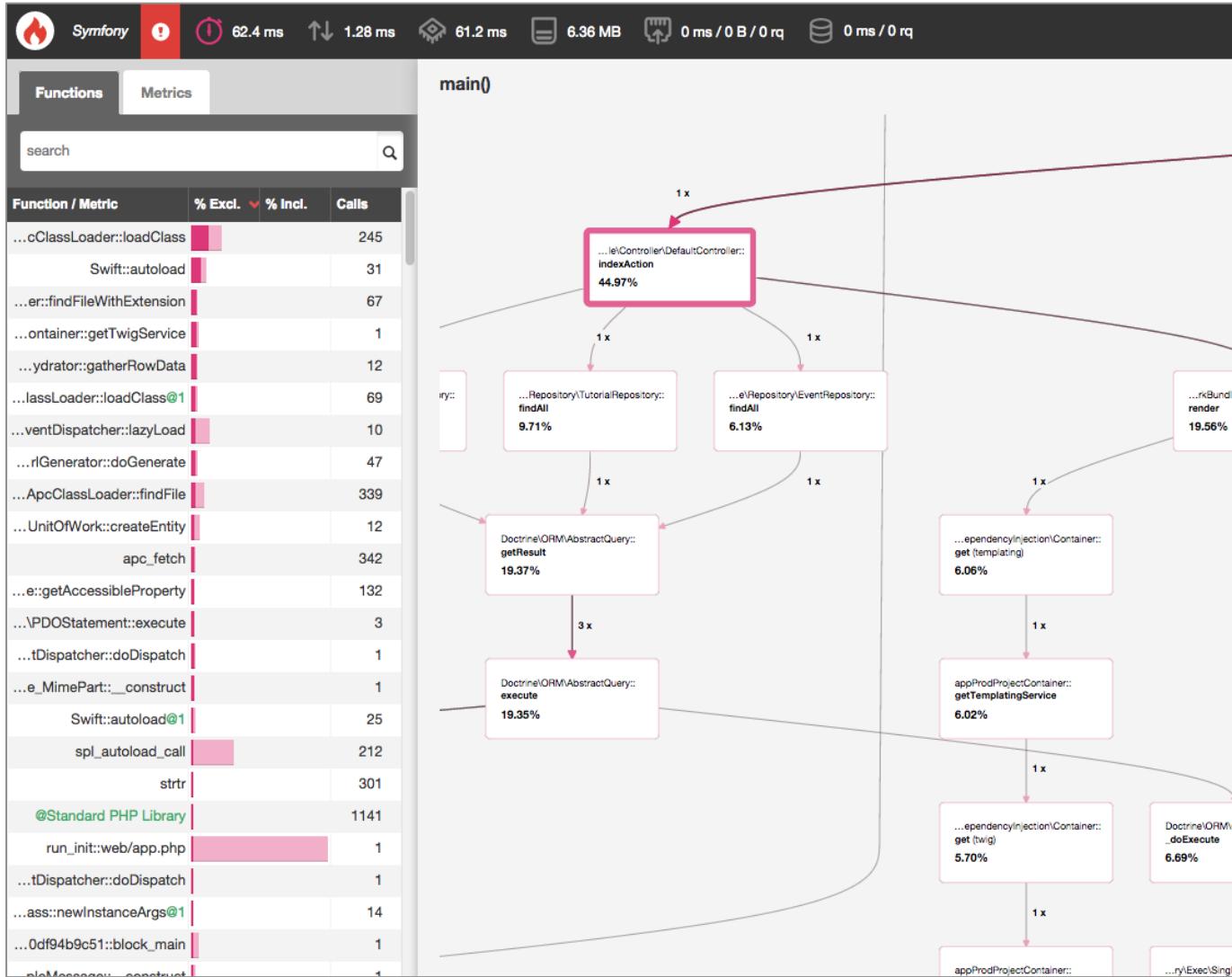
All problems are displayed on context and they provide detailed solutions.

It works for any PHP application, not only Symfony.

SensioLabs Insight helps you contain the technical debt of your projects.

# Blackfire

[blackfire.io](https://blackfire.io)



It analyzes the performance of your application to find bottlenecks.

It provides useful metrics for CPU, memory, I/O, SQL queries, HTTP requests, etc.

It works for any PHP application, not only Symfony.

Create faster applications with Blackfire.

# SensioLabs Consulting

- We develop **proof of concept applications** to evaluate Symfony for your product.
- We **coach your team** and accompany your company through the development.
- We deliver **expert missions** to help you solve specific problems in your development.
- We help you **migrate** your legacy PHP applications.

Our full list of services: [sensiolabs.com/en/packaged-solutions](http://sensiolabs.com/en/packaged-solutions)

# SensioLabs Training

- We provide **general Symfony training** for all levels (Getting Started, Mastering and Hacking) and **specific Symfony training** for testing, performance and internals.
- We provide special Symfony training for your **developers**, including web development and Twig integration.
- We provide training for other **PHP** technologies such as Doctrine and Drupal.

Our full list of courses: [training.sensiolabs.com](http://training.sensiolabs.com)

# SensioLabs Training Department

## Address

92-98 Boulevard Victor Hugo  
92 115 Clichy Cedex  
France

## Phone

+33 1 40 99 82 05

## Email

[training@sensiolabs.com](mailto:training@sensiolabs.com)

[training.sensiolabs.com](http://training.sensiolabs.com)

**SensioLabs**

