Getting started with Symfony3

SF3C1

SensioLabs

License

What is Symfony?

SensioLabs

The Symfony Project

The Symfony Project

=

components

+

framework

Symfony Components

Symfony is a **reusable** set of **standalone**, **decoupled**, and cohesive **PHP 5.5 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
- ClassLoader
- Config
- Console
- CssSelector
- Debug
- DependencyInjection
- Dotenv
- DomCrawler
- EventDispatcher
- ExpressionLanguage
- Filesystem
- Finder
- Form
- Guard
- HttpFoundation
- HttpKernel
- Icu
- Intl
- Ldap
- Locale
- Lock
- OptionsResolver
- Process
- PropertyAccess
- PropertyInfo
- Routing
- Security
- Serializer
- Stopwatch
- Templating
- Translation
- Validator
- VarDumper
- Workflow
- Yaml
- PHPUnit Bridge
- Polyfill APCu
- Polyfill PHP
- Polyfill PHP
- Polyfill PHP
- Polyfill PHP
- Polyfill PHP
- 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

Symfony releases

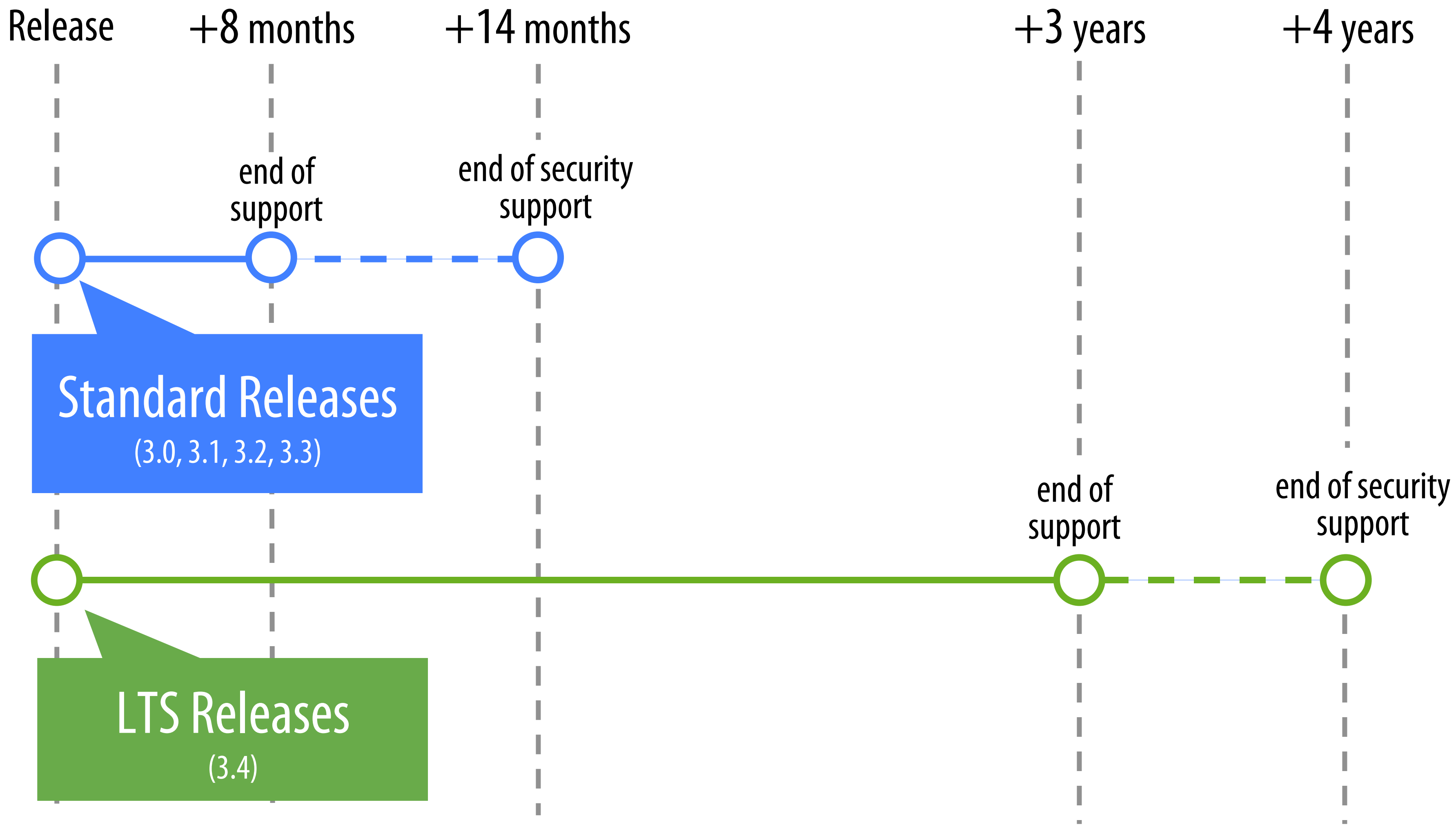
- **Patch versions** (X.Y.1, X.Y.2, X.Y.3, etc.)
  released monthly
- **Minor versions** (X.1.0, X.2.0, X.3.0, X.4.0)
  released twice a year (May and November)
  each major version releases 4 minor versions
- **Major versions** (3.0.0, 4.0.0, 5.0.0, etc.)
  released every two years

TIP Subscribe for free to receive email notifications when minor and major versions are released and/or deprecated: symfony.com/roadmap

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

eclipse

NetBeans

PhpStorm

+ Symfony Plugin

the **most popular** option for Symfony developers

Symfony Resources

Helpful Resources

- Official documentation
  - symfony.com/doc
- Official support channels
  - symfony.com/support
- Report issues or ask for new features
  - github.com/symfony/symfony

Resources to stay updated about Symfony

- Official Symfony Blog (symfony.com/blog)

  News, announcements and "New in Symfony" posts

- Community Events (symfony.com/events)

  Meetups, conferences, hackathons, etc.

- Twitter

  @symfony @symfony_live

  @symfonydocs @symfonycon

Installing Symfony

SensioLabs

Best-practice

The **Symfony Installer** is the only recommended method to install Symfony.

Installing the Symfony Installer

The Symfony Installer

- It's a tiny PHP 5.4+ application.
- It has to be installed only once.
- It works on Linux, macOS and Windows.

Installing the Installer on Linux / Mac

```
$ sudo curl -LsS \
  https://symfony.com/installer \
  -o /usr/local/bin/symfony

$ sudo chmod a+x \
  /usr/local/bin/symfony
```

Installing the Installer on Windows

```
c:\> php -r \
 "readfile('https://symfony.com/installer');" \
 > symfony
c:\> move symfony c:\projects

c:\> cd c:\projects
c:\projects\> php symfony
```

TIP If your Windows system doesn't support reading from HTTPS URLs, use http://symfony.com instead.

Updating the Symfony Installer

```
# Linux, Mac
$ symfony self-update

# Windows
c:\projects\> php symfony self-update
```

Creating a new Symfony project

Create a project with the latest Symfony version

```
# Linux, Mac
$ symfony new my-project

# Windows
c:\> php symfony new my-project
```

Create a project based on a Symfony branch

```
# Linux, Mac
$ symfony new my-project 3.0


# Windows
c:\> php symfony new my-project 3.0
```

Create a project based on a Symfony version

```
# Linux, Mac
$ symfony new my-project 3.1.3


# Windows
c:\> php symfony new my-project 3.1.3
```

Create a project based on the latest LTS version

```
# Linux, Mac
$ symfony new my-project lts

# Windows
c:\> php symfony new my-project lts
```

Check the installed Symfony version

Display the installed Symfony version

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

  Symfony version 3.1.0
  - app/dev/debug
```

Installing Symfony without the installer

Installing Symfony without the installer

- Symfony can also be installed via **Composer.**
- The result will be almost the same, but Composer is much **slower.**

Create a project using the latest Symfony version

```
$ composer \
  create-project \
  symfony/framework-standard-edition \
  my-project/
```

Create a project based on a Symfony branch

```
$ composer \
  create-project \
  symfony/framework-standard-edition \
  my-project/ \
  3.0.*
```

Display the installed Symfony version

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

  Symfony version 3.1.0
  - app/dev/debug
```

Check if your system is ready for Symfony

```
$ cd my-project/
$ php bin/symfony_requirements

  Symfony requirements check
  ...
```

Composer

Composer

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

Official website: getcomposer.org

Best-practice

Composer should be installed globally in your system.

Installing Composer on Linux / Mac

```
$ curl -sS \
  https://getcomposer.org \
  /installer | php

$ mv composer.phar \
  /usr/local/bin/composer
```

More detailed installation instructions: getcomposer.org/download

Installing Composer on Windows

Download and install the executable file **Composer-Setup.exe** that can be downloaded from [getcomposer.org]

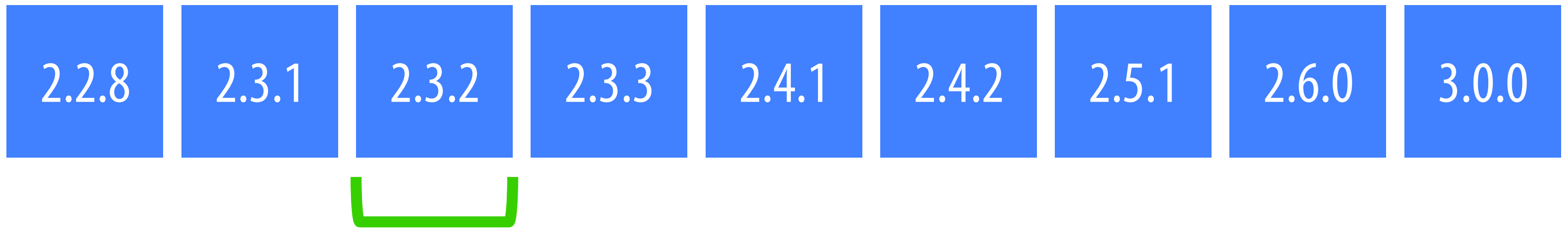
Updating Composer to the latest version

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

Composer configuration files

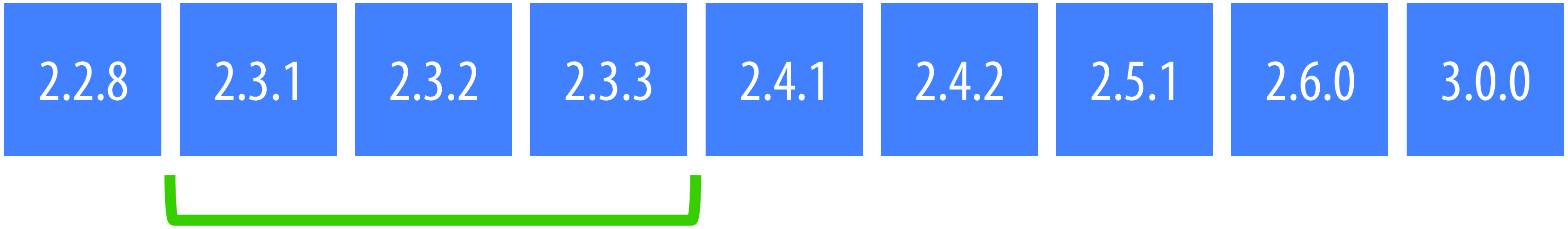
- **composer.json**
  - The dependencies + **approximate versions** that the project wants to be installed.
- **composer.lock**
  - The dependencies + **exact versions** that were installed after resolving all the dependencies.

Composer dependencies in practice



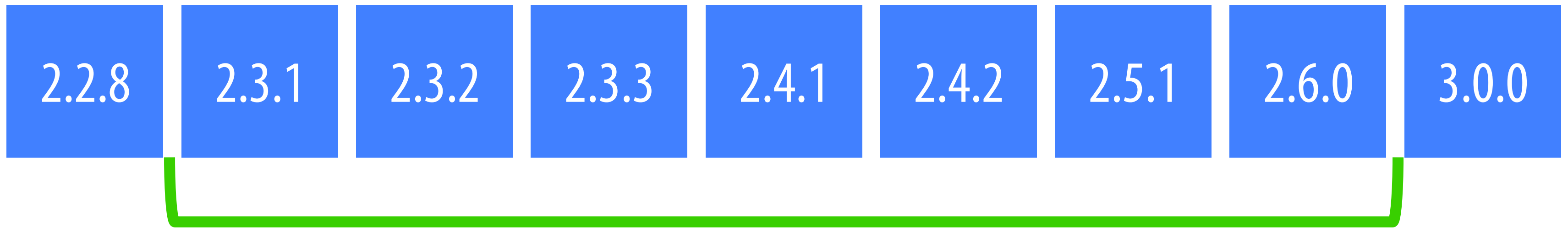
"symfony/symfony": "**2.3.2**"

Composer dependencies in practice



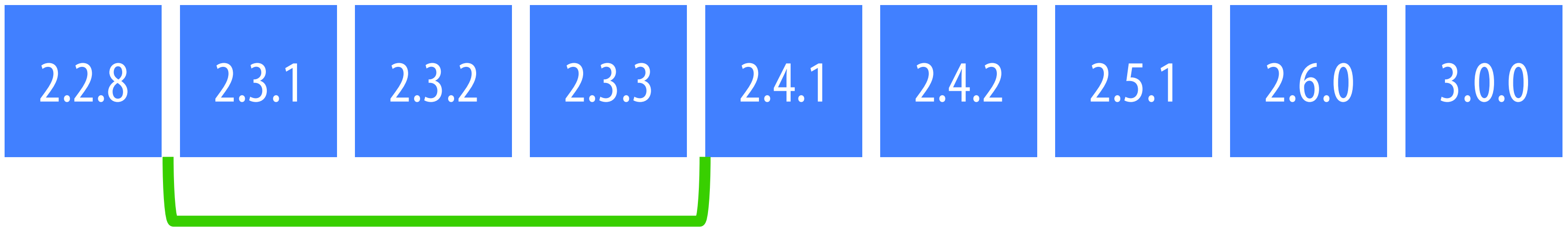
```
"symfony/symfony": "2.3.*"
```

Composer dependencies in practice



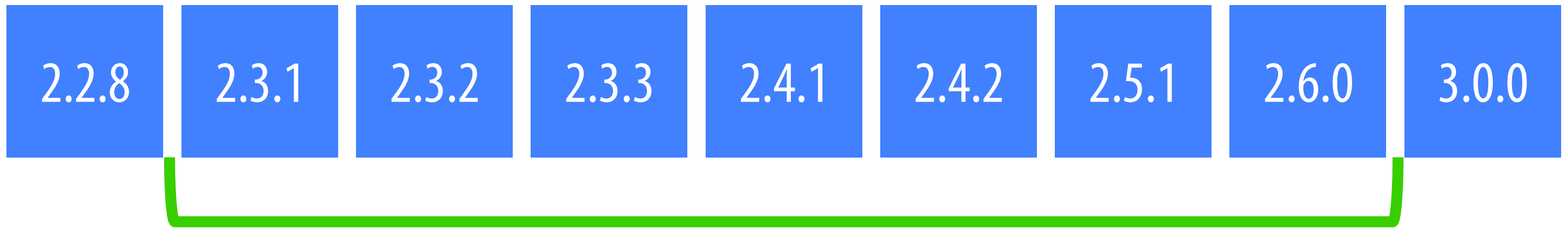
"symfony/symfony": "**~2.3**"

Composer dependencies in practice



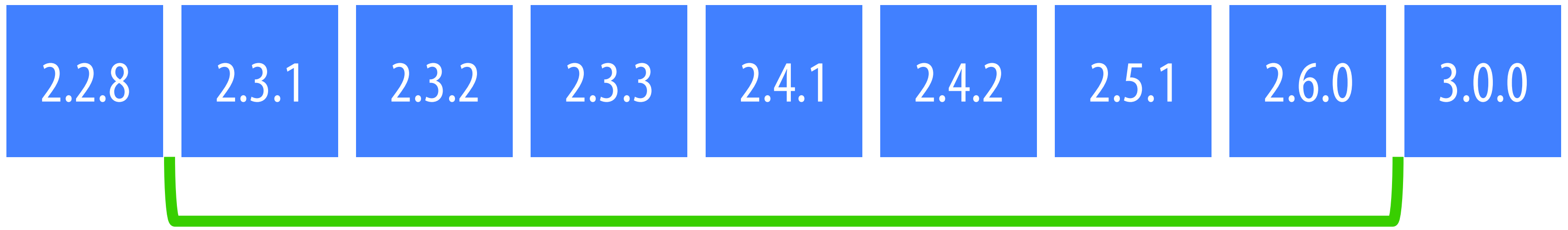
"symfony/symfony": "~2.3.1"

Composer dependencies in practice



"symfony/symfony": "^2.3"

Composer dependencies in practice



"symfony/symfony": "^2.3.1"

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 new bundle to a Symfony project

```
$ cd my-project/

$ composer require
  doctrine/doctrine-fixtures-bundle
```

Then, follow the bundle instructions to enable it, configure it, load its routes (if needed), install its assets (if needed), etc.

Adding a new 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.

Anatomy of a Symfony3 project

SensioLabs

Architecture

Overview of the directory hierarchy

```
<your-project>
├─ app/
├─ bin/
├─ src/
├─ tests/
├─ var/
├─ vendor/
└─ web/
```

The app/ directory

```
<your-project>
└ app/
  ├ autoload.php
  ├ AppKernel.php
  ├ AppCache.php
  ├ config/
  └ Resources/
```

The **application directory** contains the main configuration files, the kernel classes as well as the application resources such as templates, translations, documentation, etc.

The var/ directory

```
<your-project>
└ var/
  ├ cache/
  ├ logs/
  └ 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>
└ src/
  ├ AppBundle/
  └ Acme/
```

The **source directory** contains the PHP code of your application, both the bundles and your own business logic libraries.

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.

Its contents are managed by **Composer**. **Don't modify any file inside this folder.**

The web/ directory

```
<your-project>
└ web/
   ├ app.php
   ├ app_dev.php
   ├ images/
   ├ css/
   ├ js/
   └ ...
```

The **web directory** contains the front controllers and the web assets.

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

Overridding the default directory structure

```php
class AppKernel extends Kernel
{
    // ...

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

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

See symfony.com/doc/current/configuration/override_dir_structure.html

Configuration

Symfony3 configuration

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

YAML configuration sample

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
  #esi:    ~
  #translator: { fallback: "%locale%" }
  secret:  "%secret%"
  charset: UTF-8
  router:  { resource: "%kernel.root_dir%/config/routing.yml" }
  form:    true

# ...
```

XML configuration sample

```xml
<!-- app/config/config.xml -->
<imports>
    <import resource="parameters.yml" />
    <import resource="security.yml" />
</imports>

<framework:config charset="UTF-8" secret="xxxxxxxxxx">
    <framework:form />
    <framework:csrf-protection />
    <framework:router resource="%kernel.root_dir%/config/
routing.xml" />
    <!-- ... -->

</framework>
```

PHP configuration sample

```php
// app/config/config.php

$container->import('parameters.yml');
$container->import('security.yml');

$container->loadFromExtension('framework', array(
    'secret'          => 'xxxxxxxxxx',
    'charset'         => 'UTF-8',
    'form'            => array(),
    'csrf-protection' => array(),
    'router'          => array('resource' =>
'%kernel.root_dir%/config/routing.php'),
    // ...
));
```

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

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

Summary of configuration formats

	Pros	Cons
Annotations	Easy to read Concise	Commented code No autocompletion Hard to debug
XML	Validation IDE autocompletion	Verbose
YAML	Hierarchical configuration Easy to read	Hard to validate No native PHP support
PHP	Flexible More expressive	No validation
INI	Concise Easy to read	Very limited syntax

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.
- Don't use the **INI** format.

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
```

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
```

Using env vars in config files

requires
Symfony 3.2

```
# app/config/config.yml
doctrine:
  dbal:
    # ...
    password: "%env(DB_PASSWORD)%"
```

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

Default values for env vars

requires
Symfony 3.2

```
# app/config/parameters.yml
parameters:
    env(DB_PASSWORD): 1234
```

The special syntax **env(...)** defines the default value to use in case the given env var is not defined. Useful for the development environment.

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**.

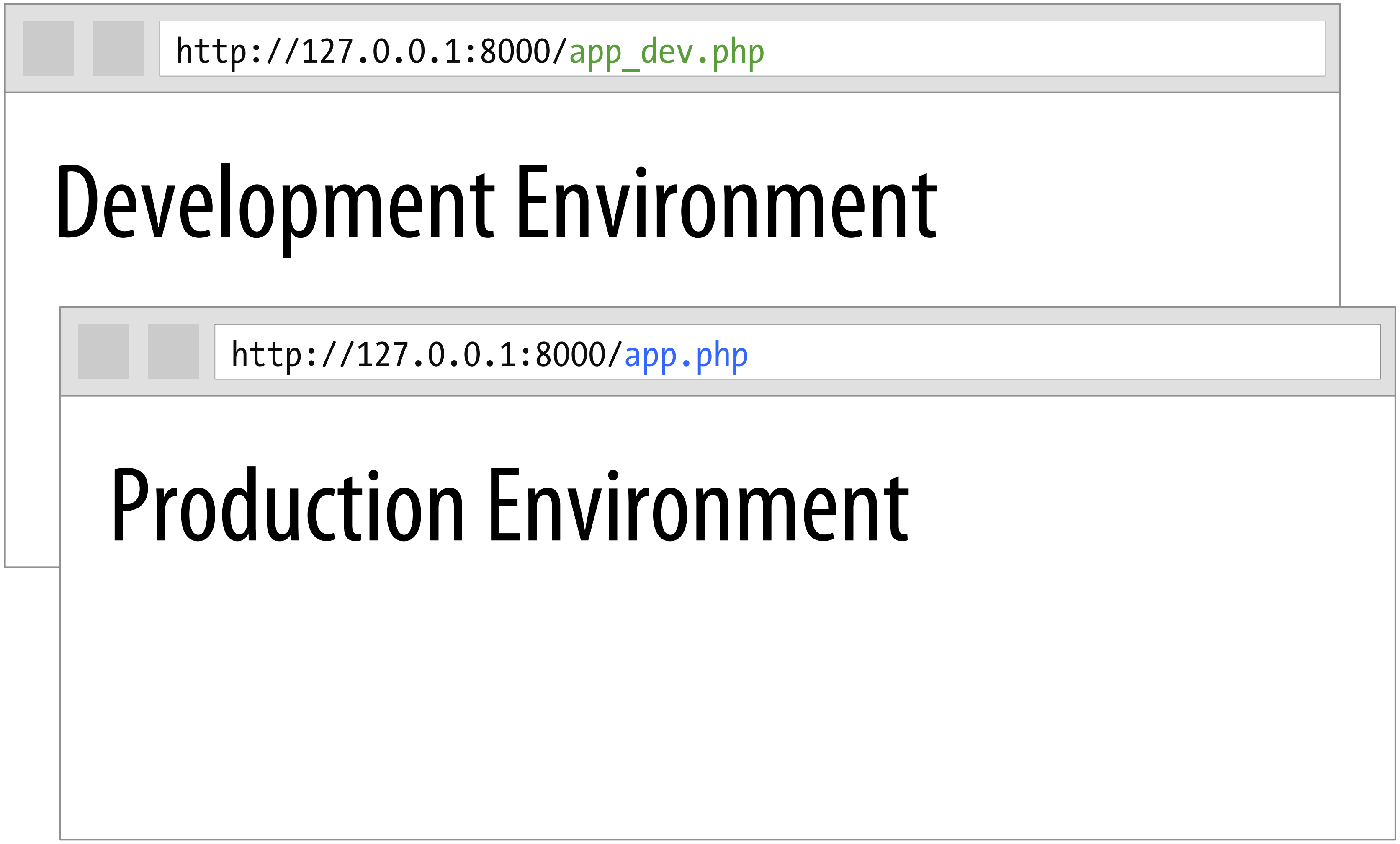
Execution environments

- 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>
 └ app/
    └ config/
       ├ config.yml
       ├ config_dev.yml        Development environment
       ├ config_prod.yml       Production environment
       ├ routing.yml
       ├ routing_dev.yml       Development environment
       └ ...
```

Front controllers select the environment



http://127.0.0.1:8000/app_dev.php

Development Environment

http://127.0.0.1:8000/app.php

Production Environment

Front controllers select the environment

```
// web/app.php
$kernel = new AppKernel('prod', false);

// web/app_dev.php
$kernel = new AppKernel('dev', true);
```

The name of the environment

Whether to enable debugging or not

Which configuration file is loaded by Symfony?

```
class AppKernel extends Kernel
{
    // ...

    public function
    registerContainerConfiguration($loader)
    {
        $loader->load(__DIR__.'/config/'
           'config_'.$this->getEnvironment().'.yml'
        );
    }
}
```

Which configuration file is loaded by Symfony?

```yaml
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

# app/config/config_prod.yml
imports:
    - { resource: config.yml }

# app/config/config.yml
framework:
    # ...

twig:
    # ...
```

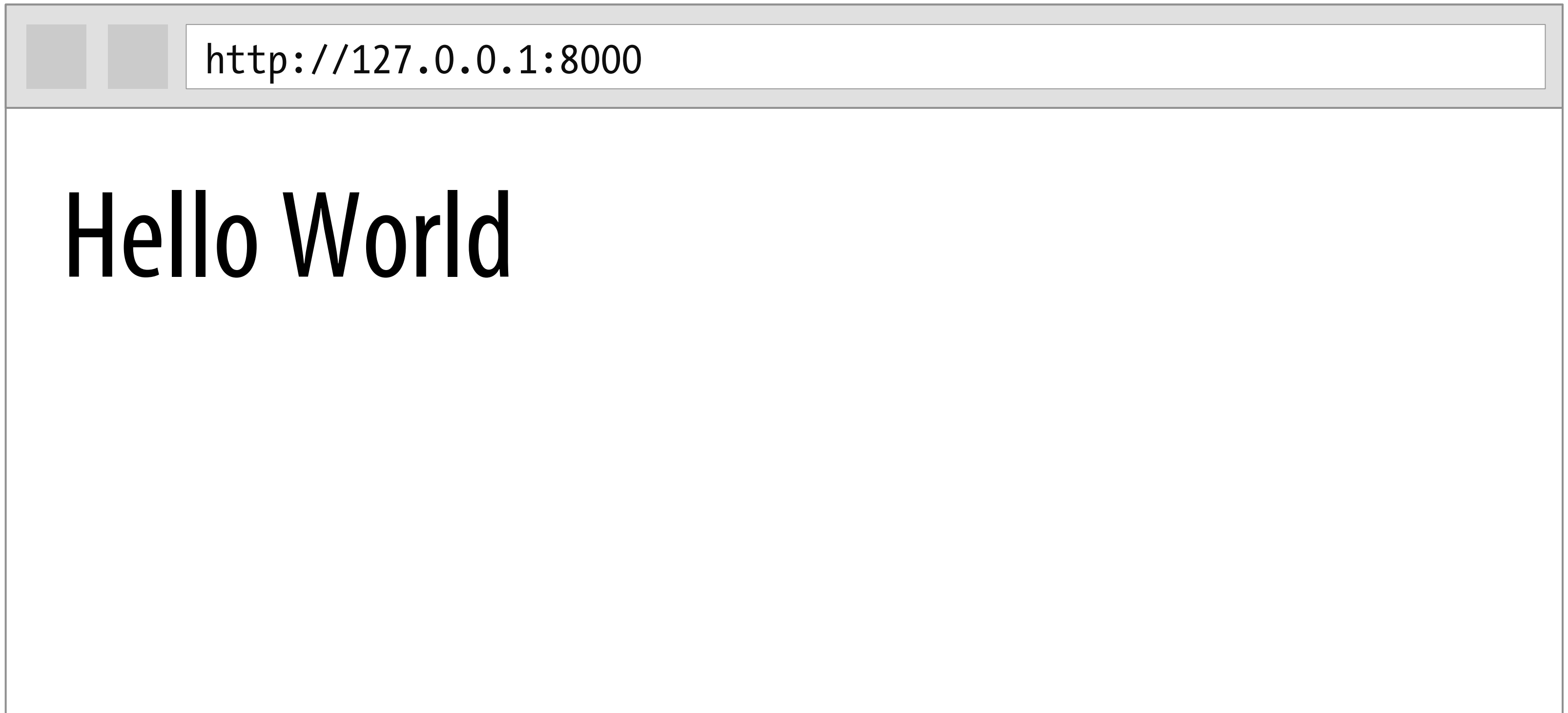
Hello Symfony World

SensioLabs

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


Hello World
```

HTTP Response

received from the server

Processing HTTP requests with raw PHP code

```php
<?php

// load and initialize any 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
<?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

```
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

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

This code shows **Hello World** when accessing the homepage of the site.

Web server configuration

Best-practice

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

Using the PHP built-in web server

requires
PHP 5.4

```
$ cd my-project/
$ php bin/console server:run

  Server running on
  http://127.0.0.1:8000
```

Using Apache Web Server

web/ is the only public directory for Symfony applications

```
<VirtualHost *:80>
    ServerName       my-project.dev
    DocumentRoot     "/projects/my-project/web"
    DirectoryIndex   app.php

    <Directory "/projects/my-project/web">
        AllowOverride None
        Allow from All
    </Directory>

    <IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteCond   %{REQUEST_FILENAME} !-f
        RewriteRule   ^(.*)$ app.php [QSA,L]
    </IfModule>
</VirtualHost>
```

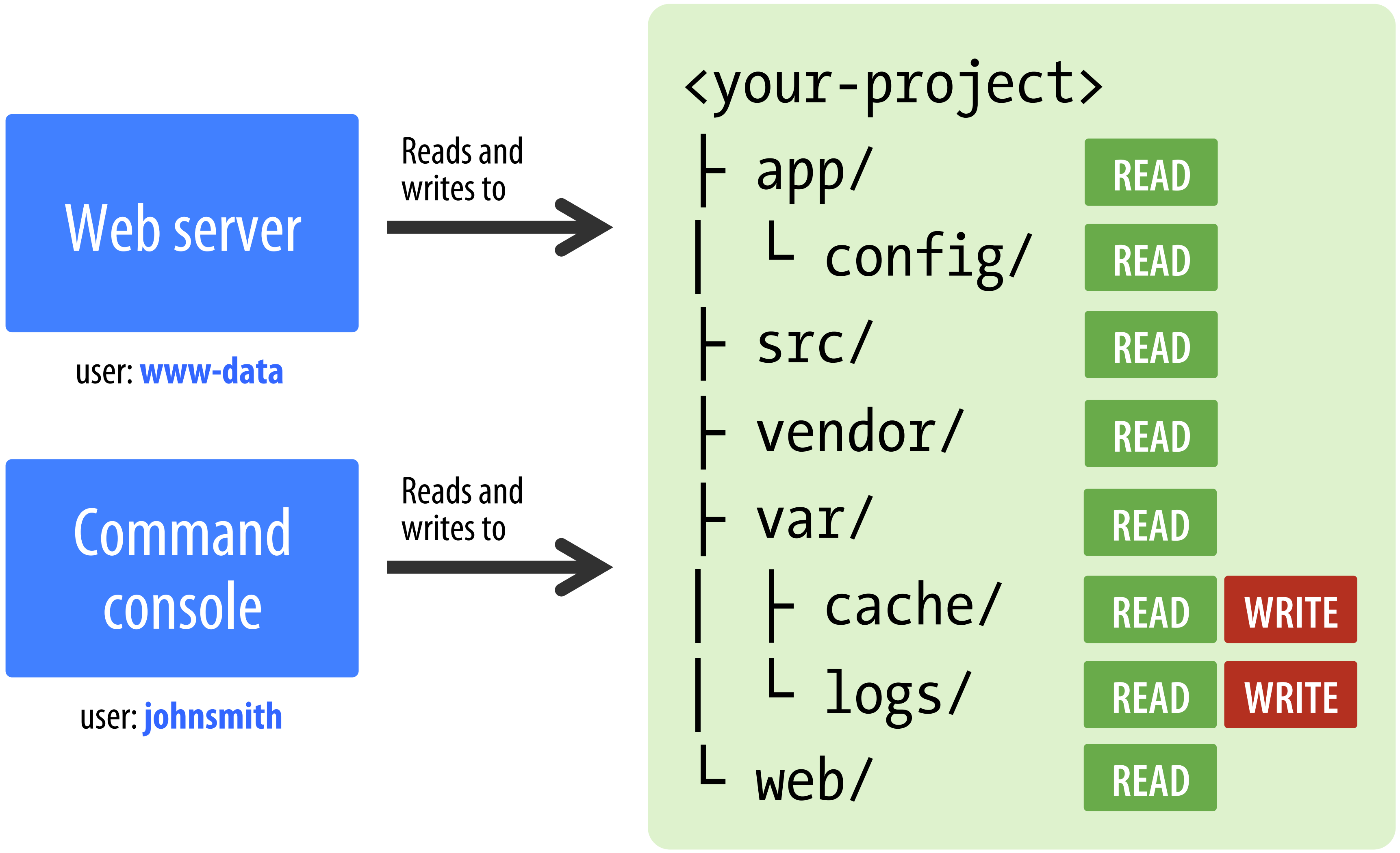
Using Nginx Web Server

web/ is the only public directory for Symfony applications

```
server {
    server_name my-project.dev;
    root /projects/my-project/web;

    location / {
        try_files $uri /app.php$is_args$args;
    }

    location ~ ^/(app|app_dev|config)\.php(/|$) {
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param HTTPS off;
    }

    error_log /var/log/nginx/project_error.log;
    access_log /var/log/nginx/project_access.log;
}
```

Setting up permissions

Understanding the permission problem



```
<your-project>
├ app/                READ
│  └ config/          READ
├ src/                READ
├ vendor/             READ
├ var/                READ
│  ├ cache/           READ  WRITE
│  └ logs/            READ  WRITE
└ web/                READ
```

Best-practice

Change the user of the web server to match the user of the command console.

Setting the user of the web server

```
// Apache
// [...]/conf/httpd.conf
User  johnsmith
Group staff

// Nginx
// [...]/nginx.conf and [...]/php-fpm.conf
user  johnsmith
group staff
```

Restart the web server after changing the value of these options.

Alternative #1: chmod

```
# delete existing cache and log contents
$ rm -rf var/cache/*
$ rm -rf var/logs/*

# fix permissions
$ sudo chmod +a "www-data allow
delete,write,append,file_inherit,directory_inherit"
var/cache var/logs

$ sudo chmod +a "`whoami` allow
delete,write,append,file_inherit,directory_inherit"
var/cache var/logs
```

Alternative #2: setfacl

```
# delete existing cache and log contents
$ rm -rf var/cache/*
$ rm -rf var/logs/*

# fix permissions
$ sudo setfacl -Rn -m u:"www-data":rwX -m
u:`whoami`:rwX var/cache var/logs

$ sudo setfacl -dRn -m u:"www-data":rwX -m
u:`whoami`:rwX var/cache var/logs
```

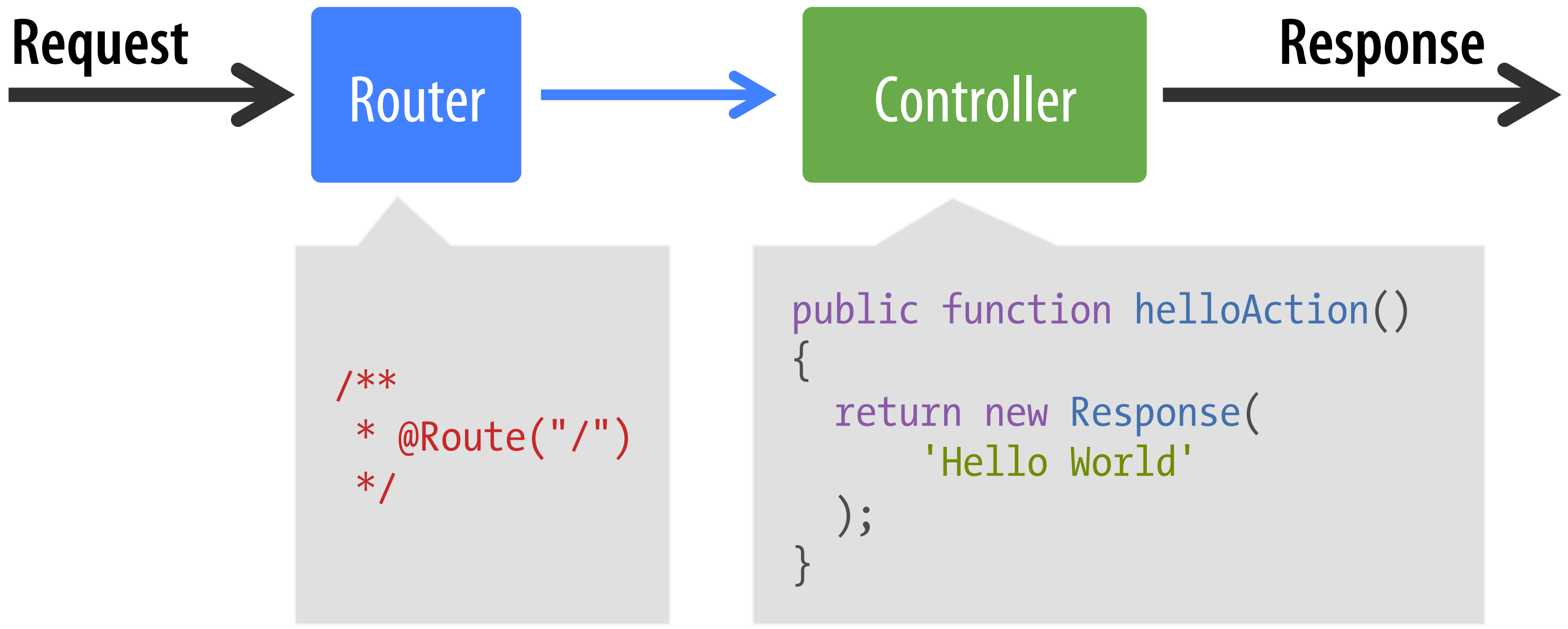
Alternative #3: Vagrant

Either use the NFS option on UNIX hosts or add the Vagrant user to the webserver's group in the Vagrantfile.

```
# using unix hosts
config.vm.synced_folder "./", "/vagrant", id:
"vagrant-root", :nfs => true


# using windows or other systems without nfs support
config.vm.synced_folder "./", "/vagrant", id:
"vagrant-root",
  owner: "vagrant",
  group: "www-data",
  mount_options: ["dmode=775,fmode=664"]
```

The Request - Response flow

The simplest Request-Response Flow



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

```
public function helloAction()
{
  return new Response(
      'Hello World'
  );
}
```

Symfony is ...

✓ A Request/Response framework.

✓ An HTTP framework.

✗ A MVC framework.

(Model-View-Controller)

Rendering a template (1 of 2)

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
~~use Symfony\Component\HttpFoundation\Response;~~

class DefaultController extends Controller
{
    /**
     * @Route("/")
     */
    public function helloAction()
    {
        ~~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)

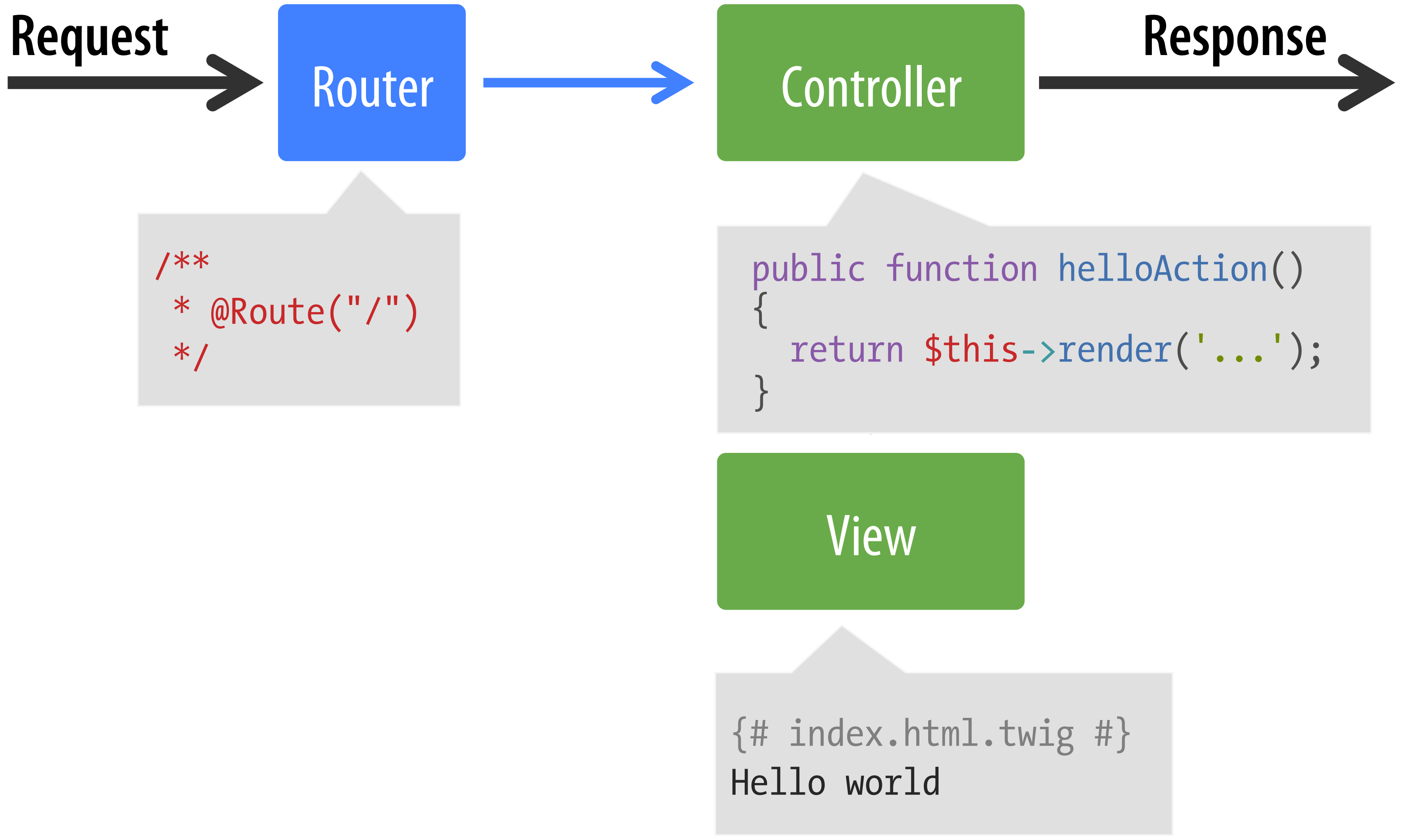
```php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
~~use Symfony\Component\HttpFoundation\Response;~~

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

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

Rendering a template (2 of 2)

```
{# app/Resources/views/index.html.twig #}
Hello world
```

The advanced Request-Response Flow

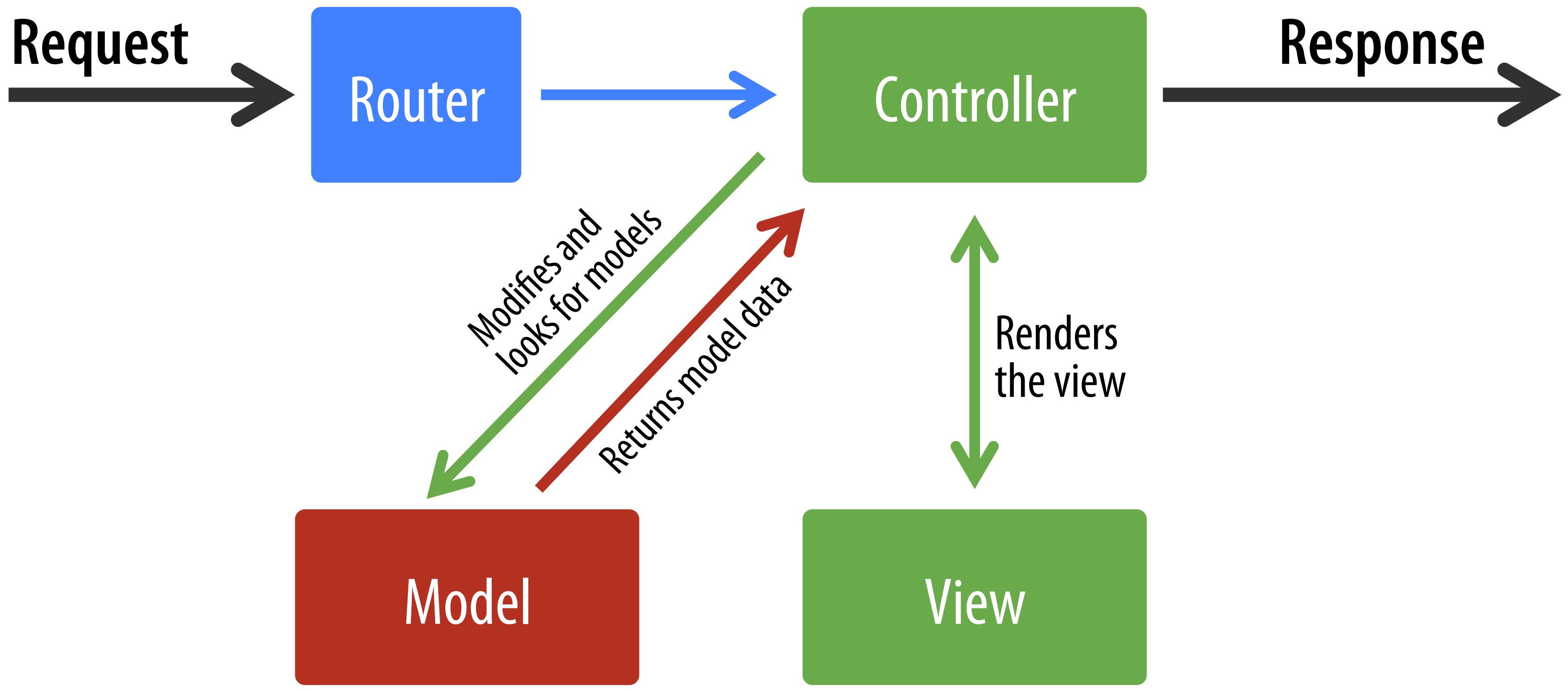


```
/**
 * @Route("/")
 */
```

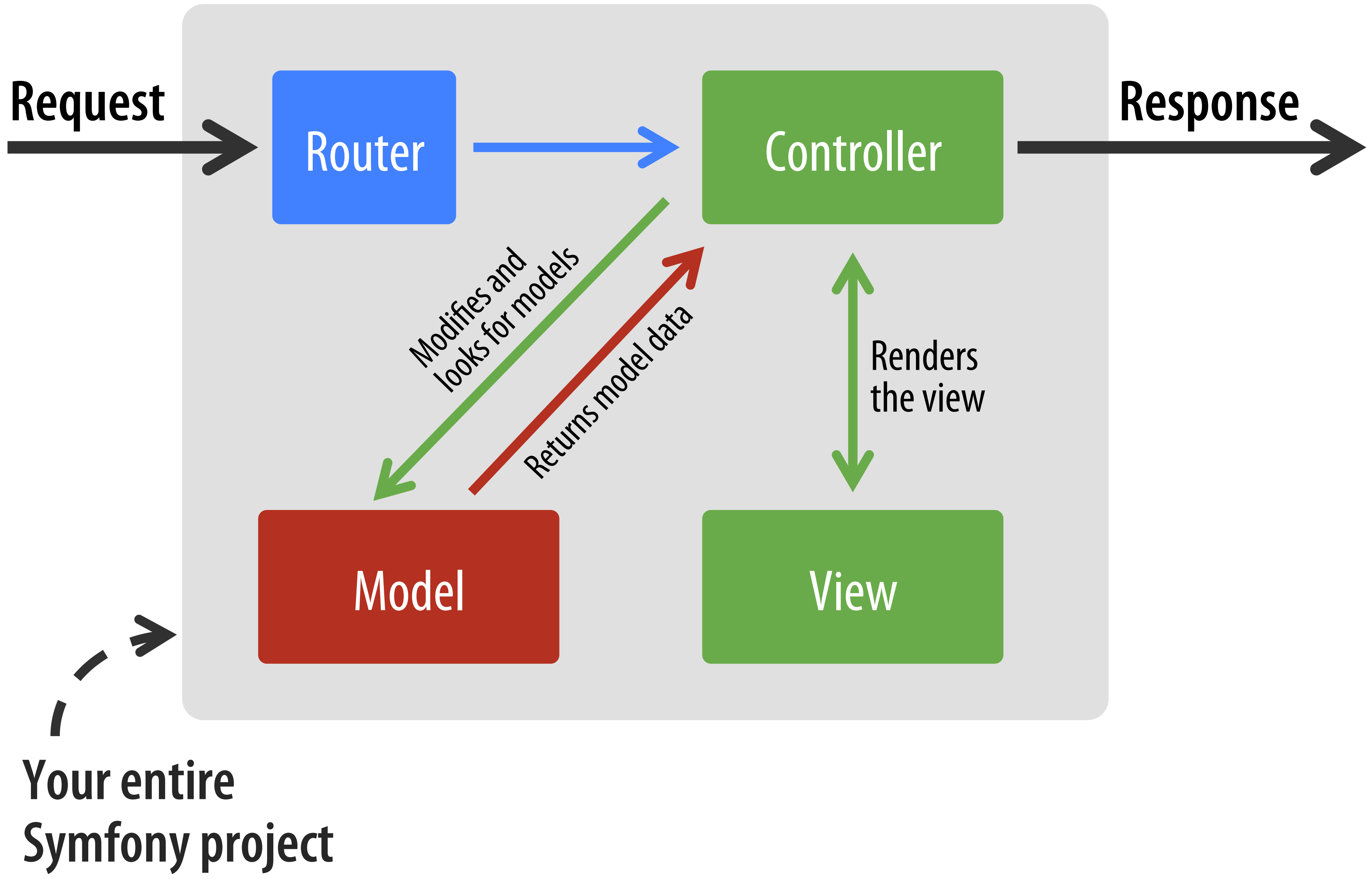
```
public function helloAction()
{
    return $this->render('...');
}
```

```
{# index.html.twig #}
Hello world
```

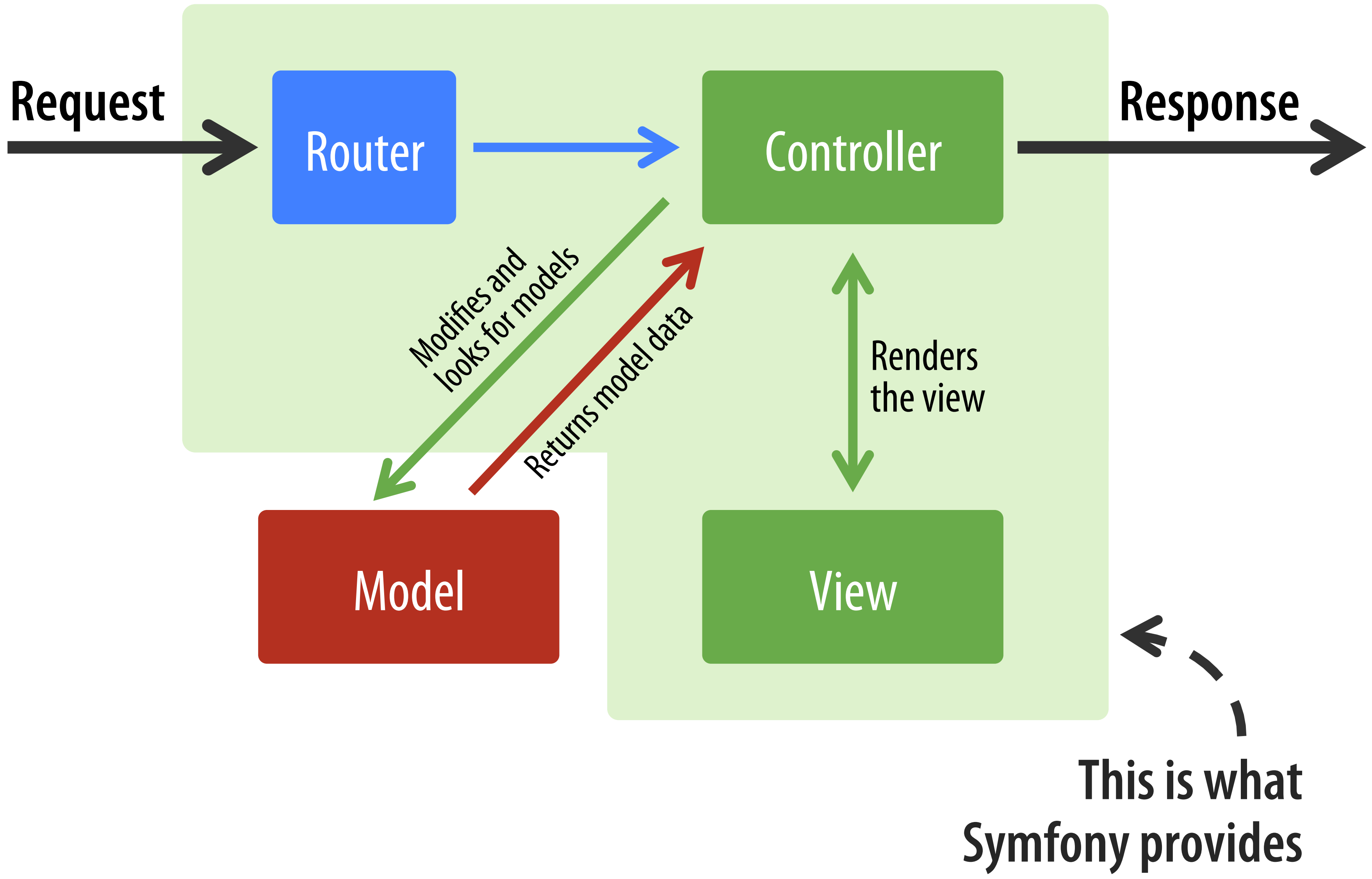
The complete Request-Response Flow



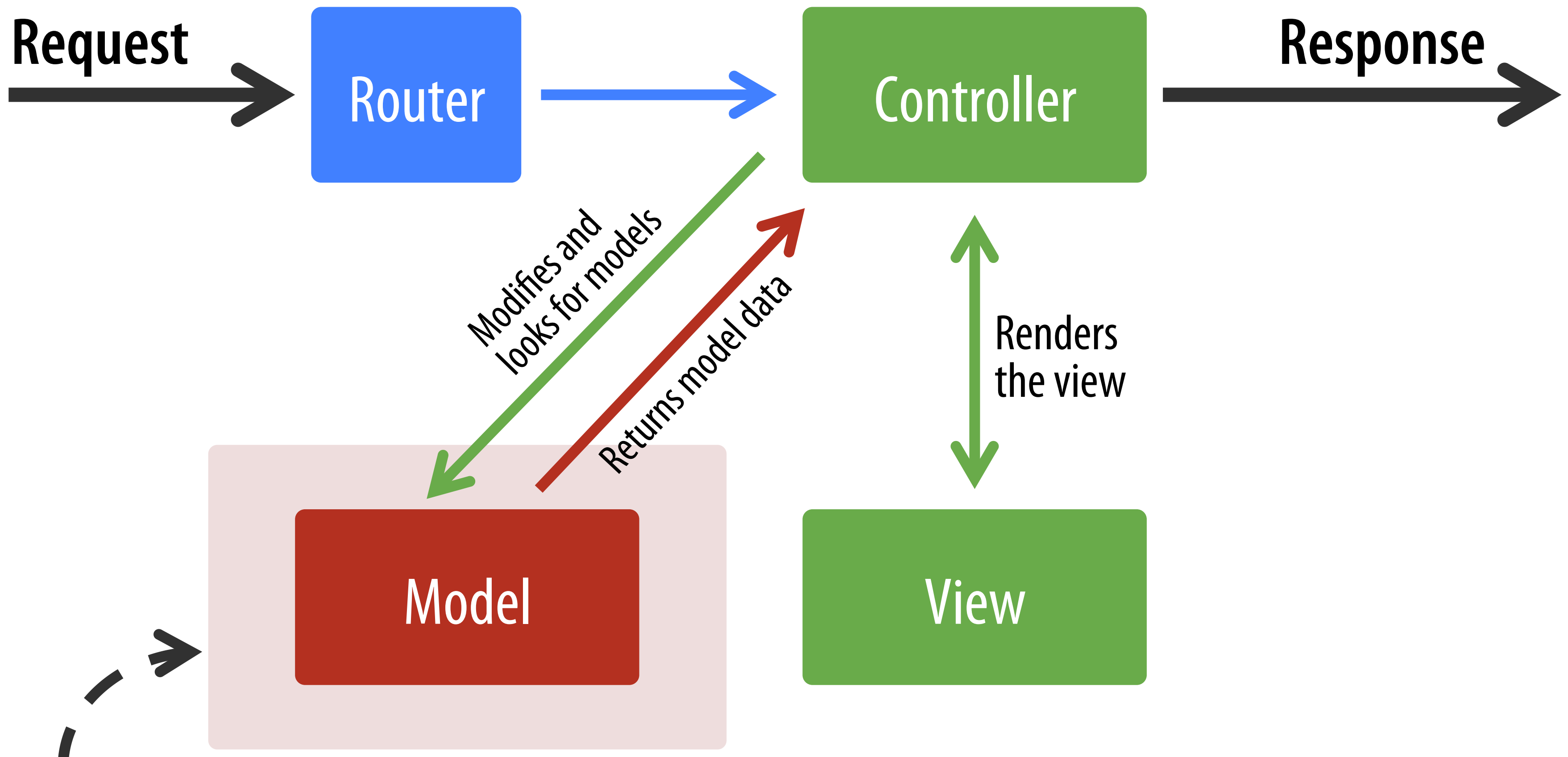
The complete Request-Response Flow



The complete Request-Response Flow



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.

A simple route example

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

A route with placeholders (variables)

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

A route with default values (1 of 2)

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

A route with default values (2 of 2)

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

A route with constraints (1 of 2)

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

A route with constraints (2 of 2)

```
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;

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

A complex route example

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;

/**
 * @Route(
 *   "/blog/{page}",
 *   defaults={"page": "1"},
 *   requirements={ "page": "\d+" },
 *   name="blog_list"
 * )
 * @Method({ "GET", "HEAD" })
 */
public function listAction($page)
{
    // ...
}
```

A YAML route example

```yaml
# app/config/routing.yml
blog_list:
    path: /blog/{page}
    defaults:
        _controller: AppBundle:Blog:list
        page: 1
    requirements:
        page: \d+
    methods: [GET, HEAD]
```

Introduction to Twig

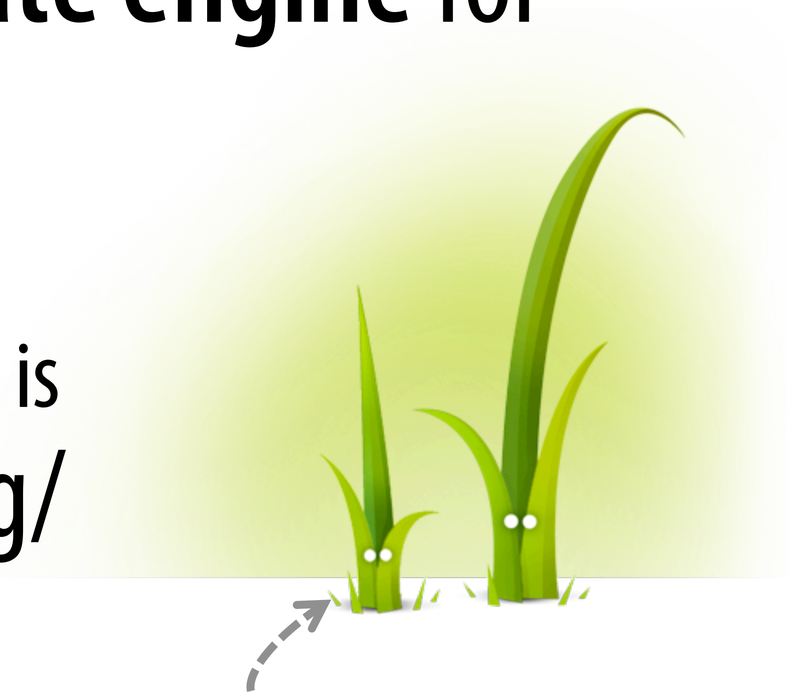
SensioLabs

What is Twig

What is Twig

Twig is a **modern template engine** for PHP.

The official website for the project is http://twig.sensiolabs.org/



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

```
# app/config/config.yml
twig:
    strict_variables: false
```

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

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

```
# app/config/config.yml
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
- slice
- sort
- split
- striptags
- title
- trim
- upper
- url_encode

Official Twig documentation: twig.sensiolabs.org/documentation

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
- block
- constant
- cycle
- date
- dump
- include
- max
- min
- parent
- random
- range
- source
- template_from_string

Official Twig documentation: twig.sensiolabs.org/documentation

Output escaping

Automatic output escaping

```
Hi {{ name }}!
```

The variable **name** is **automatically escaped** if it contains a string

Automatic output escaping

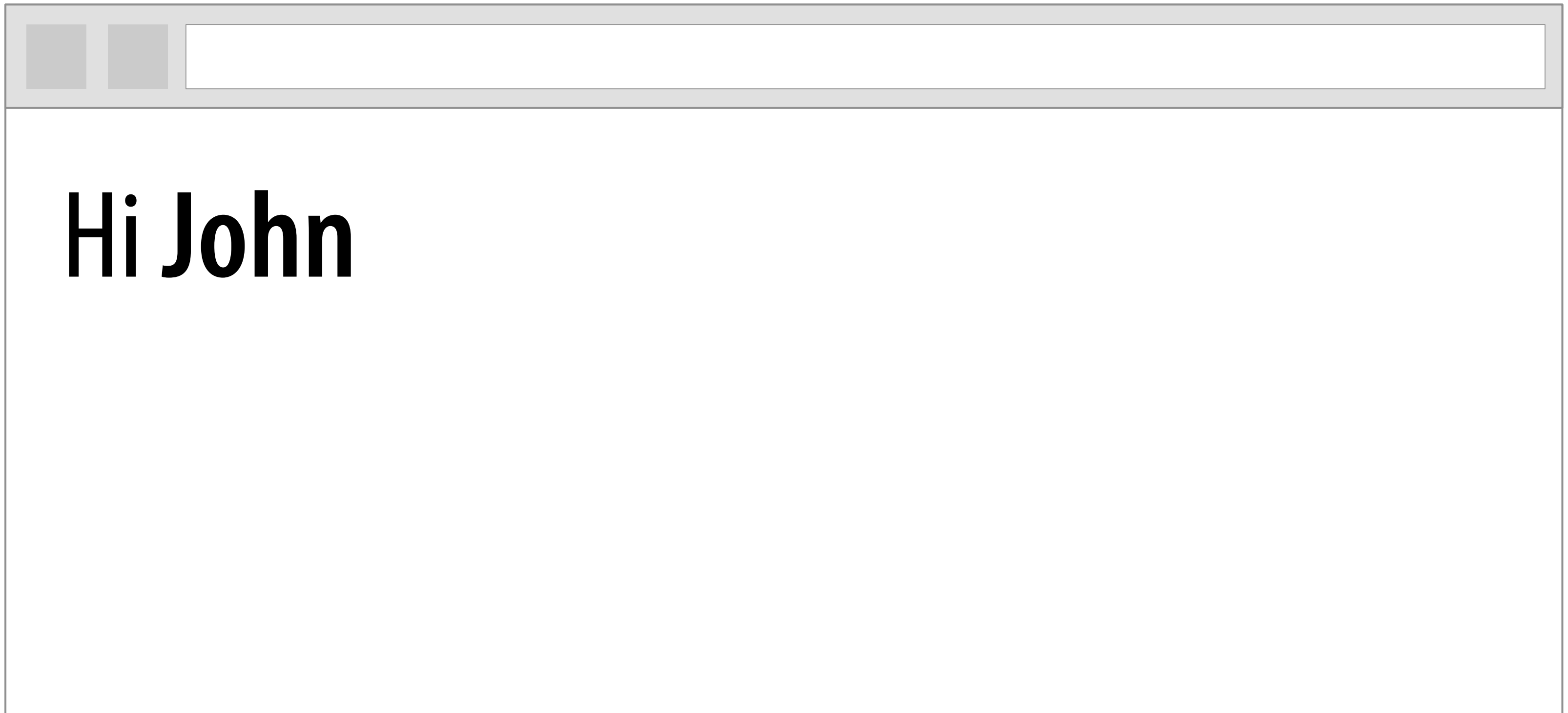
```
Hi {{ name }}
```

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

Automatic output escaping

Expected output

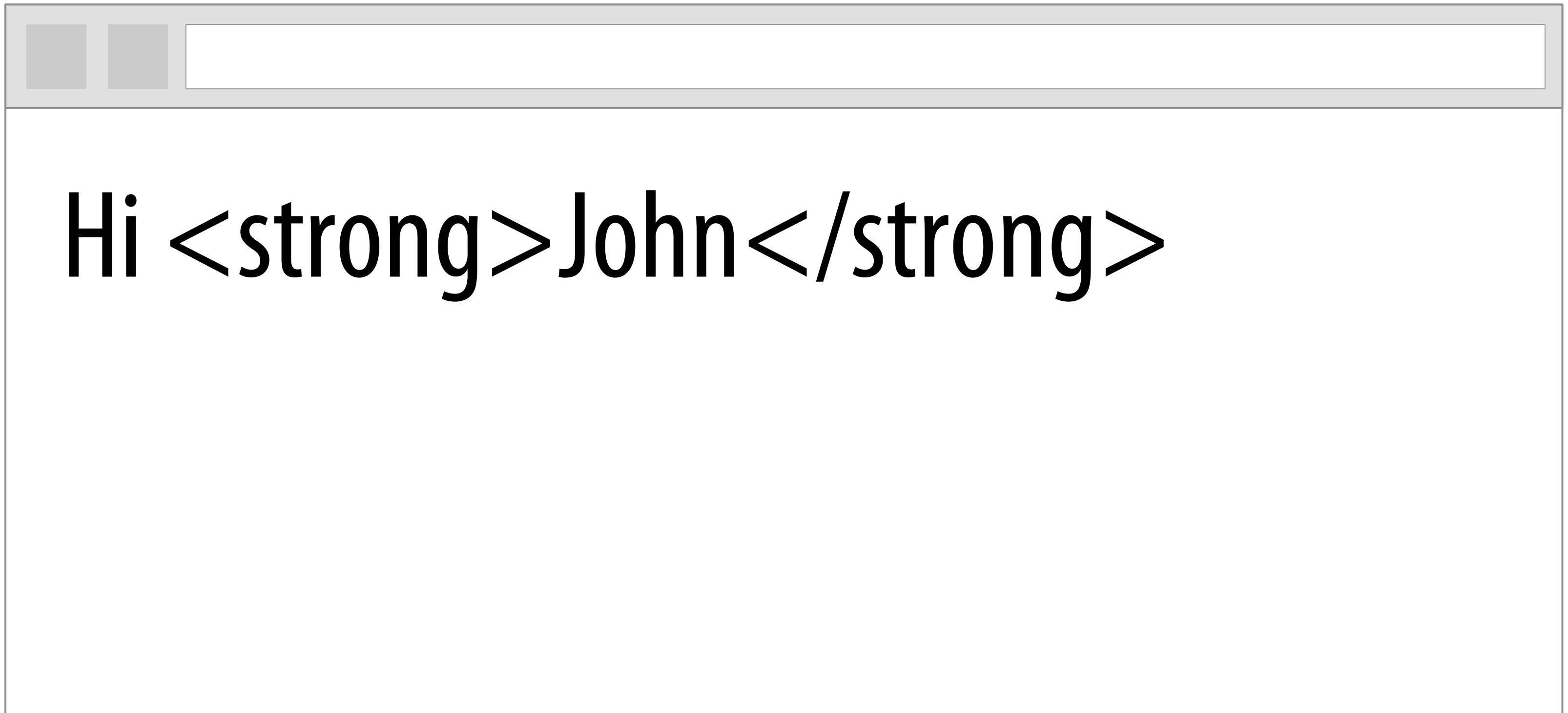
```
Hi <strong>John</strong>
```


Hi **John**

Automatic output escaping

Real output

```
Hi &lt;strong&gt;John&lt;/strong&gt;
```



Hi <strong>John</strong>

Control structures

Comparison of control structures

Twig

if	else
elseif	for

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 filters the collection before iterating over it with the **for** statement.

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 %}
  <p>
    Hello <strong>{{ name }}</strong>!
  </p>
{% endspaceless %}
```

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

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.

```
{# app/Resources/views/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

```
{# app/Resources/views/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 %}Blog{% 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   'Blog' %}

{% block body %}
    <h1>Latest posts</h1>
    {{ include('posts.twig') }}
{% endblock body %}
```

Reusing the content of any block

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

{% block body_id 'blog_index' %}
{% block title   'Blog' %}

{% block body %}
    <h1>{{ block('title') }}</h1>
    {{ include('posts.twig') }}
{% endblock body %}
```

Parent templates can define default contents

```
{# app/Resources/views/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 %}

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

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

Removes the default parent value

Overrides the default parent value

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 every variable that exists in the template

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

```
/* AppBundle:Default: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

SensioLabs

Global variables

Defining global variables

```yaml
# app/config/config.yml
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

Linking to web assets stored in web/ directory

```
<img alt="Symfony!"
  src="{{ asset('images/logo.png') }}"
/>

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

Assets must be located in the **web**/ directory.

Linking to web assets stored in bundles

```
<img alt="Symfony!" src="{{ asset(
  'bundles/app/images/logo.png'
) }}"/>


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

Assets must be located in the **Resources/public/** directory of the bundle.

Installing web assets defined by bundles

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

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

Defining asset version

```
# app/config/config.yml
framework:
    # ...
    assets:
        version: "v=2"
```

```
<img src="{{ asset('logo.png') }}" />
```

Template

```
<img src="/logo.png?v=2" />
```

Output

Defining asset base URL

```
# app/config/config.yml
framework:
    # ...
    assets:
        base_urls:
            - 'http://static.example.com'
```

```
<img src="{{ asset('logo.png') }}" />
```

Template

```
<img src="http://static.example.com/
logo.png" />
```

Output

Filters and functions

Controller functions

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

{{ render(controller(
  'AppBundle:Article:latest', { 'max': 3 }
)) }}

{{ render_esi(controller(
  'AppBundle:Article:latest', { 'max': 3 }
)) }}
```

Documentation and examples: https://symfony.com/doc/current/reference/twig_reference.html

Form functions

```
{% form_theme form '@App/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:
https://symfony.com/doc/current/reference/twig_reference.html

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:
https://symfony.com/doc/current/reference/twig_reference.html

Translation filters

```
{{ message|trans }}

{{ message|transchoice(5) }}

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

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

Documentation and examples: https://symfony.com/doc/current/reference/twig_reference.html

Commands

Twig linter

Checks if the syntax of the Twig templates is valid.

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

OK in src/Blogger/BlogBundle/Resources/views/Blog/show.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Comment/_comments.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Comment/_form.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Comment/new.html.twig
OK in src/Blogger/BlogBundle/Resources/views/layout.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Page/_sidebar.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Page/about.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Page/contact.html.twig
OK in src/Blogger/BlogBundle/Resources/views/Page/contactEmail.txt.twig
OK in src/Blogger/BlogBundle/Resources/views/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.
- In Symfony apps, third-party bundles templates are overridden in app/Resources/NameOfTheBundle/views/

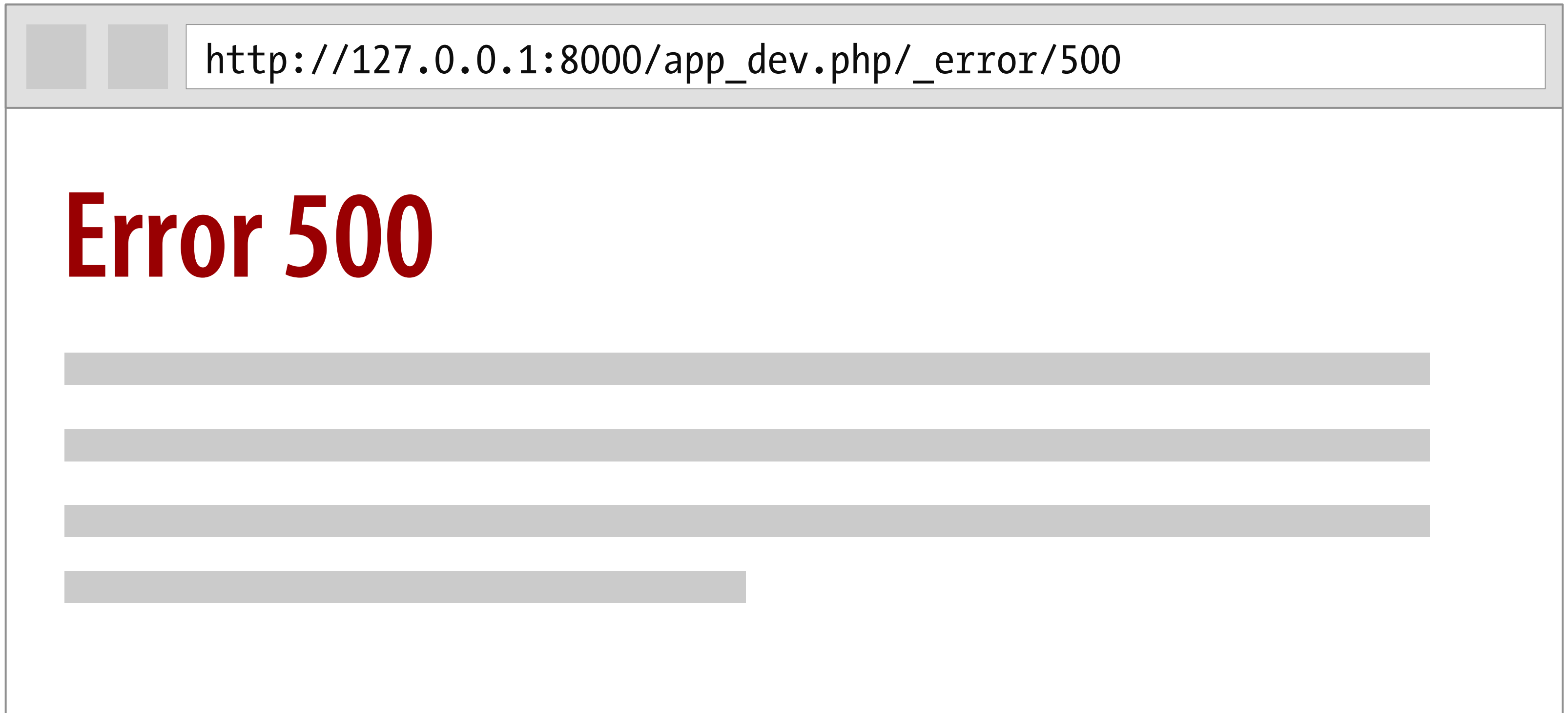
Overridding error templates

```
app/
└─ Resources/
    └─ TwigBundle/
        └─ views/
            └─ 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

SensioLabs

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
- ISO-3166-1 alpha-2 defines country codes
  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

	Language	Country
fr_FR	French	France
fr_BE	French	Belgium

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

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

Enable and configure translation

```yaml
# app/config/config.yml
framework:
    translator:
        fallbacks:  ['fr', 'en']
```

By default, **translation** is disabled to avoid any impact in the application performance.

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

Define the default locale

```
# app/config/config.yml
framework:
    default_locale: 'en'
```

This is the **default locale** used when no locale is explicitly defined by the given user. You can only define one default locale which is applied to all users.

Complete translation configuration

```yaml
# app/config/config.yml
framework:
    default_locale: 'en'
    translator:
        fallbacks: ['fr', 'en']
```

Step 2.

Extract content strings

Translating contents outside templates

```
public function indexAction()
{
    $title = $this->get('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**.

Translating template contents

```
{% trans %}
  Contact us
{% endtrans %}
```

Use **Twig tags** to translate large blocks of static contents.

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

Use **Twig filters** to translate variables and expressions.

Main difference between filters and tags

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

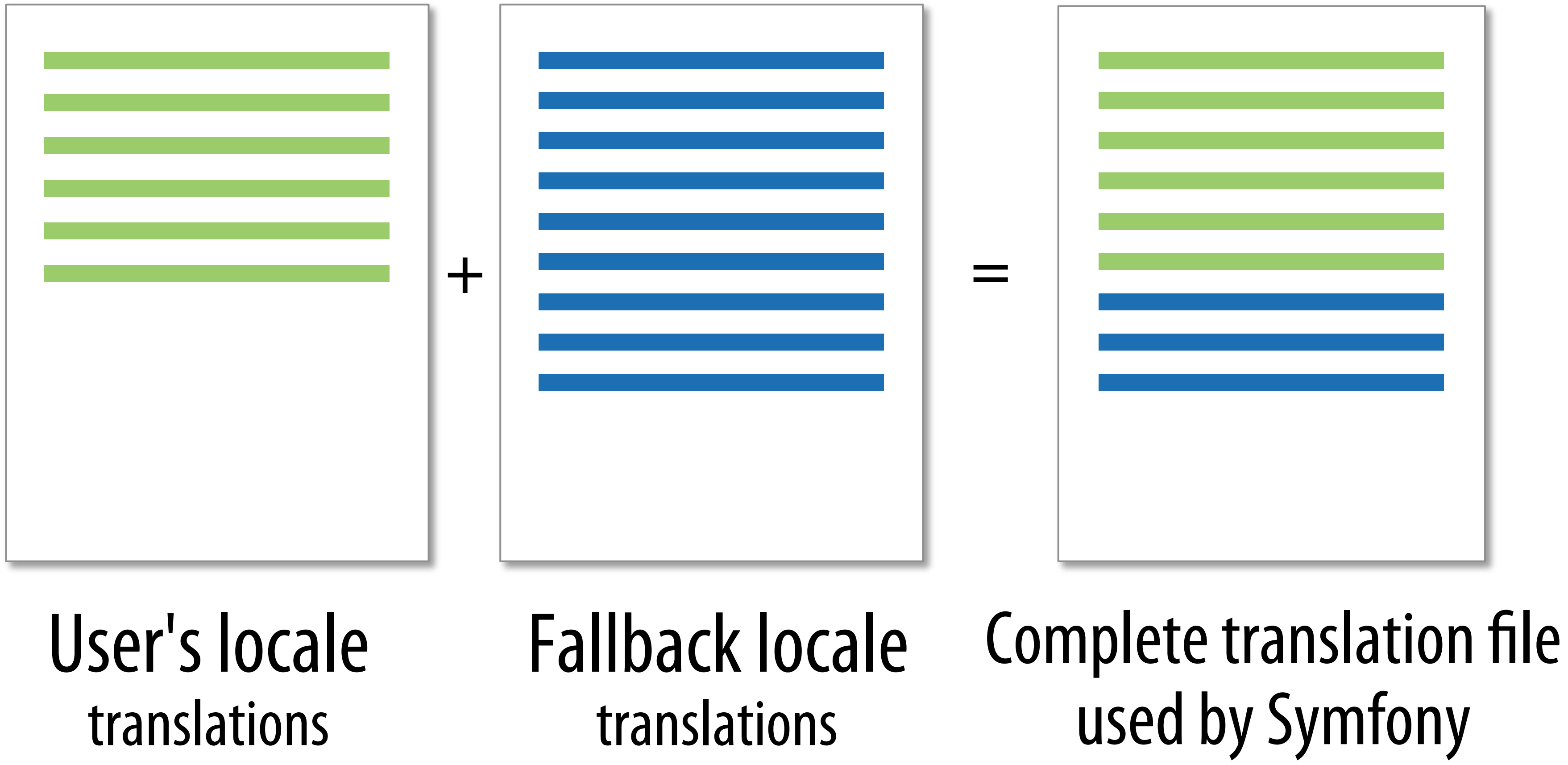
```
OUTPUT <h1>Contactez-nous</h1>
```

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

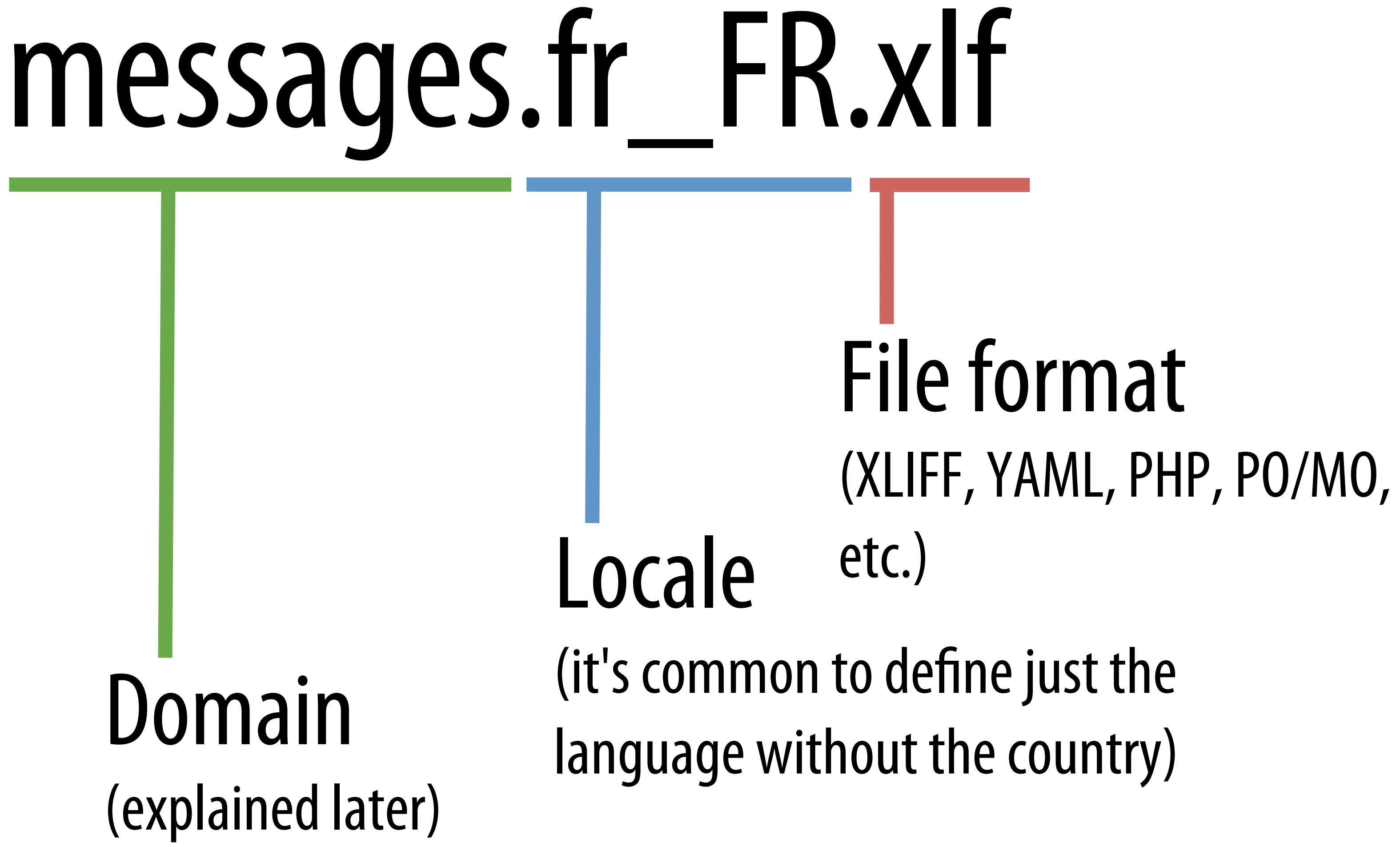
```
OUTPUT &lt;h1&gt;Contactez-nous&lt;/h1&gt;
```

Step 3.
Create translation files

How does Symfony get the translation



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/
├─ app
│   └─ Resources
│       ├─ translations/
│       │   └─ messages.fr.xlf
│       └─ AcmeBlogBundle/
│           └─ translations/
│               └─ messages.fr.xlf
├─ ...
└─ src
    └─ Acme/
        └─ BlogBundle/
            └─ Resources
                └─ translations/
                    └─ messages.fr.xlf
```

Symfony applies an **overriding mechanism** to select the catalogue to use.

This allows to override any bundle translation, including third-party bundles.

Translation files priority

`app/Resources/translations/messages.fr.xlf`

HIGHEST priority. It **OVERRIDES** any other catalogue with the same name and locale, regardless of where it's defined originally.

`app/Resources/AcmeBlogBundle/translations/messages.fr.xlf`

MEDIUM priority. It **OVERRIDES** any catalogue with the same name and locale defined by a bundle with the same name as this directory.

`src/Acme/BlogBundle/Resources/translations/messages.fr.xlf`

LOWEST priority. It can be **OVERRIDDEN** by any catalogue with the same name and locale defined in the **app/** directory.

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

```xml
<!-- app/Resources/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

```yaml
# app/Resources/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

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

public function indexAction(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

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

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

_locale (with a leading underscode) 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/cookbook/session/locale_sticky_session.html

Forcing the translation locale in the controller

```
public function indexAction()
{
  $title = $this->get('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

```
{{ 'Contact us'|trans(
  { }, 'messages', '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.

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

```php
$this->get('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({ }, '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).

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

```
public function indexAction()
{
  $title = $this->get('translator')->trans(
    'Hello %name%',
    array('%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

```xml
<!-- app/Resources/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

```yaml
# app/Resources/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

```php
public function indexAction()
{
  $title = $this->get('translator')->transChoice(
    'There is one product left.|There are %count%
products left.',
    10
  );
}
```

Message alternatives are separated with a pipe (|)

This is the value considered to decide which message to pick (singular or plural).

Translating plural messages in templates

```
{% transchoice 10 %}
  There is one product left.|There are %count%
products left.
{% endtranschoice %}


{{ 'There is one product left.|There are %count%
products left.'|transchoice(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: https://en.wikipedia.org/wiki/Interval_(mathematics)#Notations_for_intervals

Full reference of trans() and transchoice()

```php
$this->get('translator')->trans(
    'Hello %name%',
    array('%name%' => 'John'),
    'admin',
    'fr_FR'
);

$this->get('translator')->transChoice(
    'There is one product left.|There are %count% products left.',
    10,
    array(),
    'admin',
    'fr_FR'
);
```

Full reference of |trans and |transchoice

```
{{ message|trans }}
{{ message|trans({'%name%': 'John'}, 'admin', 'fr') }}

{{ message|transchoice(10) }}
{{ message|transchoice(10, {'%name%': 'John'},
                                    'admin', 'fr') }}
```

Full reference of {% trans %} and {% transchoice %}

```
{% trans with {'%name%': 'John'} from 'admin' into 'fr_FR' %}
  Hello %name%
{% endtrans %}

{% transchoice count with {'%name%': 'John'} from 'admin'
   into 'fr_FR' %}
  'There is one product left.|There are %count% products left.'
{% endtranschoice %}
```

Form and database translation

Translating form validation messages

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

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

```xml
<!-- 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 Stof**DoctrineExtensions**Bundle
  https://github.com/stof/StofDoctrineExtensionsBundle
- Use **Translatable** extension.

Creating / updating translation files automatically

Log missing translations

```
# app/config/config.yml
translator:
    logging: true
```

```
# app/logs/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 AppBundle

+----------+--------------------+-------------------------+
| State(s) | Id                 | Message Preview (fr)    |
+----------+--------------------+-------------------------+
|          | title.post_list    | Liste des articles      |
|          | action.show        | Voir                    |
|          | action.edit        | Editer                  |
|          | action.create_post | Créer un nouvel article |
+----------+--------------------+-------------------------+

Legend:
 x Missing message
 o Unused message
 = Same as the fallback message
```

Create the translation catalogues

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

Generating "en" translation files for "app/ folder"
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 "app/ folder"
Parsing templates
Loading translation files
Writing files
```

```
# app/Resources/translations/messages.en.yml
title.post_list: __title.post_list
action.show: __action.show
action.edit: __action.edit
action.create_post: __action.create_post
```

Create the translation catalogues

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

Generating "en" translation files for "app/ folder"
Parsing templates
Loading translation files
Writing files
```

```
# app/Resources/translations/messages.en.yml
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=xlf


Generating "en" translation files for "app/ folder"
Parsing templates
Loading translation files
Writing files
```

```
<!-- app/Resources/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

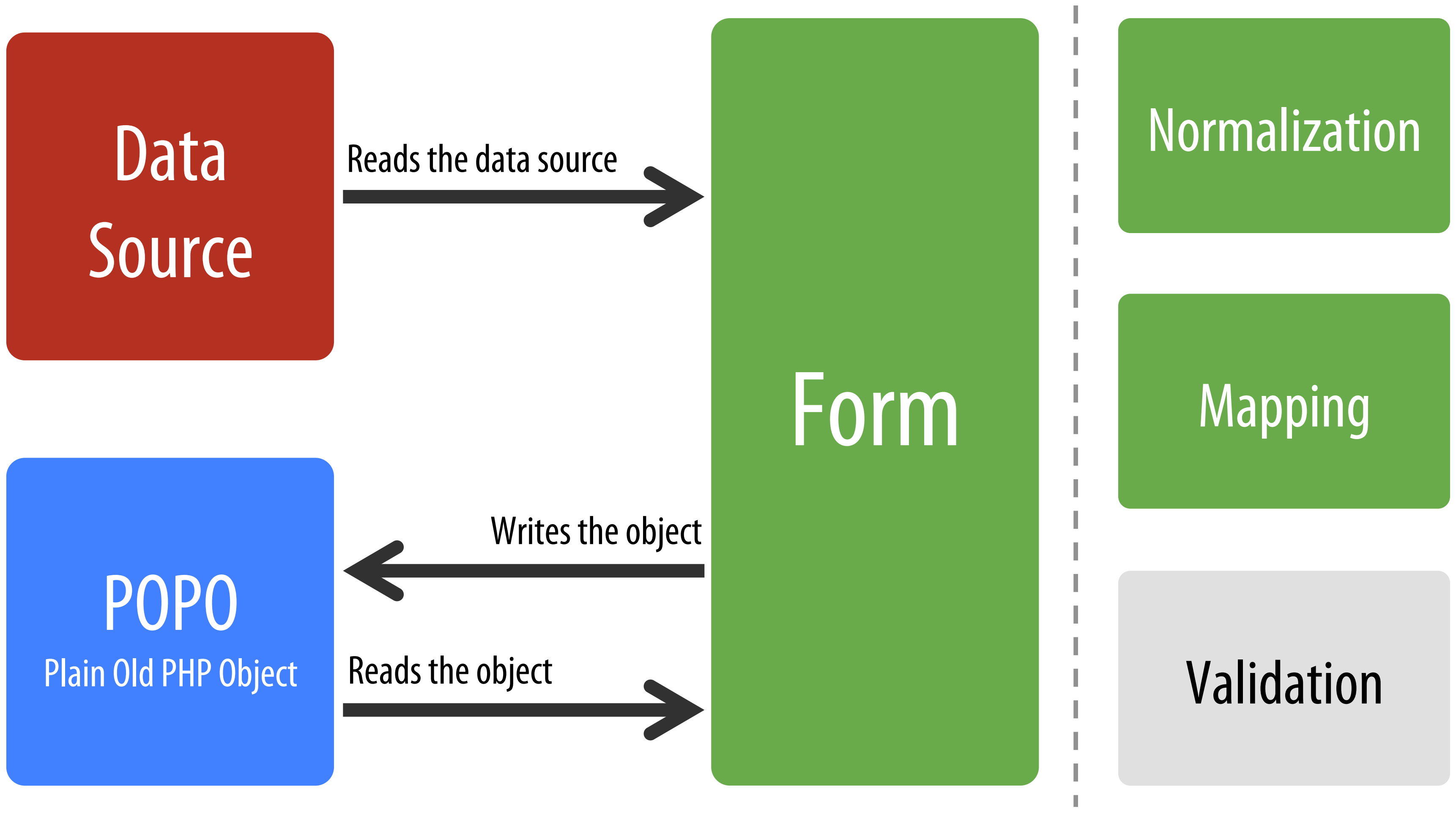
```
$ php bin/console translation:update en --force --prefix=new_
                  --output-format=xlf AppBundle


Generating "en" translation files for "AppBundle"
Parsing templates
Loading translation files
Writing files
```

```
<!-- src/AppBundle/Resources/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>
```

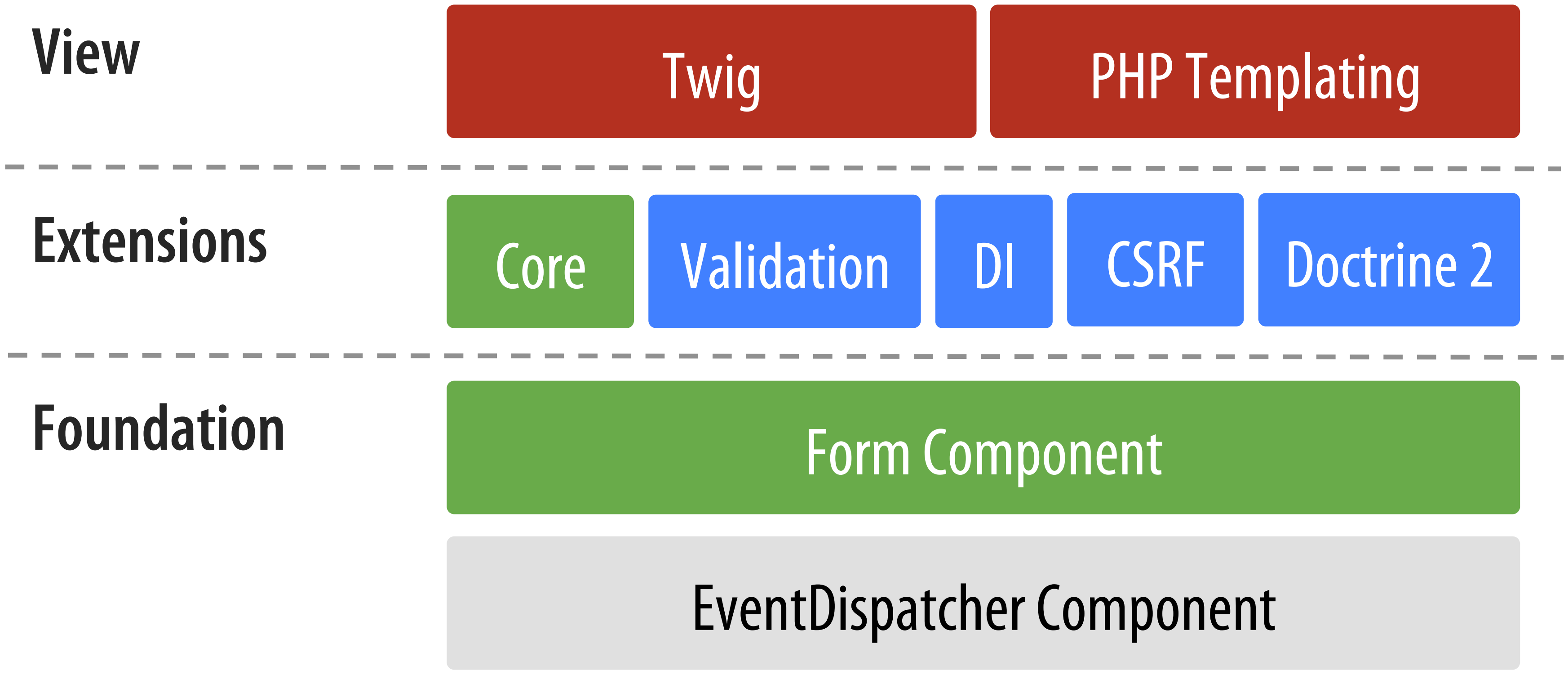
Introduction to Forms

Basic concepts

Symfony Form Component worflow



Symfony Form Component architecture



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

The domain object

Creating the domain object class

```
namespace AppBundle\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

```php
class ProductController extends Controller
{
    public function newAction()
    {
        $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', array(
            'form' => $form->createView()
        ));
    }
}
```

CAUTION it's not recommended to build forms in the controller unless they are trivial.

Building the form in the controller

```php
class ProductController extends Controller
{
    public function newAction()
    {
        $product = new Product();
1       $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', array(
            '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 Controller
{
    public function newAction()
    {
        $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', array(
            '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

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

```
class ProductController extends Controller
{
    public function newAction()
    {
        $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', array(
            'form' => $form->createView()
        ));
    }
}
```

Building the form in the controller

```
class ProductController extends Controller
{
    public function newAction()
    {
        $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', array(
            '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 Controller
{
    public function newAction()
    {
        $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', array(
            '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

```php
// src/AppBundle/Form/ProductType.php
namespace AppBundle\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, array('currency' => 'USD'))
        ;
    }
}
```

Building the form in a separate class

```
// src/AppBundle/Form/ProductType.php
namespace AppBundle\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, array('currency' => 'USD'))
        ;
    }
}
```

1. All custom types must extend from AbstractType and implement the buildForm() method.

Building the form in a separate class

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

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

class ProductType extends AbstractType 1
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
      2 $builder
            ->add('name')
            ->add('price', MoneyType::class, array('currency' => 'USD'))
        ;
    }
}
```

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

```php
use AppBundle\Entity\Product;
use AppBundle\Form\ProductType;

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

    $form = $this->createForm(ProductType::class, $product);
    $form->add('send', SubmitType::class);

    // ...
}
```

Using a form class in the controller

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

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

    $form = $this->createForm(ProductType::class, $product);
    $form->add('send', SubmitType::class);

    // ...
}
```

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 AppBundle\Entity\Product;
use AppBundle\Form\ProductType;

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

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

    // ...
}
```

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.

Using a form class in the controller

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

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

2   $form = $this->createForm(ProductType::class, $product);
3   $form->add('send', SubmitType::class);

    // ...
}
```

3. Optionally you can manipulate the Form object to add or remove any of its fields.

Tip: it's common to add buttons programatically in the controller instead of the form class.

Built-in Symfony Form Types

Text Fields

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

Choice Fields

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

Date and Time Fields

- DateType
- DateTimeType
- TimeType
- BirthdayType

Other Fields

- CheckboxType
- FileType
- RadioType

Field Groups

- CollectionType
- RepeatedType

Hidden Fields

- HiddenType

Buttons

- ButtonType
- ResetType
- SubmitType

Full details: http://symfony.com/doc/current/reference/forms/types.html

Adding validation constraints

Validation constraints as annotations

```
namespace AppBundle\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

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

Validation constraints as an XML file

```xml
<!-- Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mapping>
    <class name= "AppBundle\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

```
Resources/
└ config/
    └ validation.yml
```

```
Resources/
└ config/
    └ validation/
        ├ Author.yml
        ├ Category.yml
        ├ Comment.yml
        └ Post.yml
```

Built-in Symfony Validation Constraints

Basic Constraints

- NotBlank
- Blank
- NotNull
- Null
- IsTrue
- IsFalse
- Type

String Constraints

- Email
- Length
- Url
- Regex
- Ip
- Uuid

Number Constraints

- Range

Comparison Constraints

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

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: https://symfony.com/doc/current/reference/constraints.html

Translating validation messages (1 of 2)

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

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

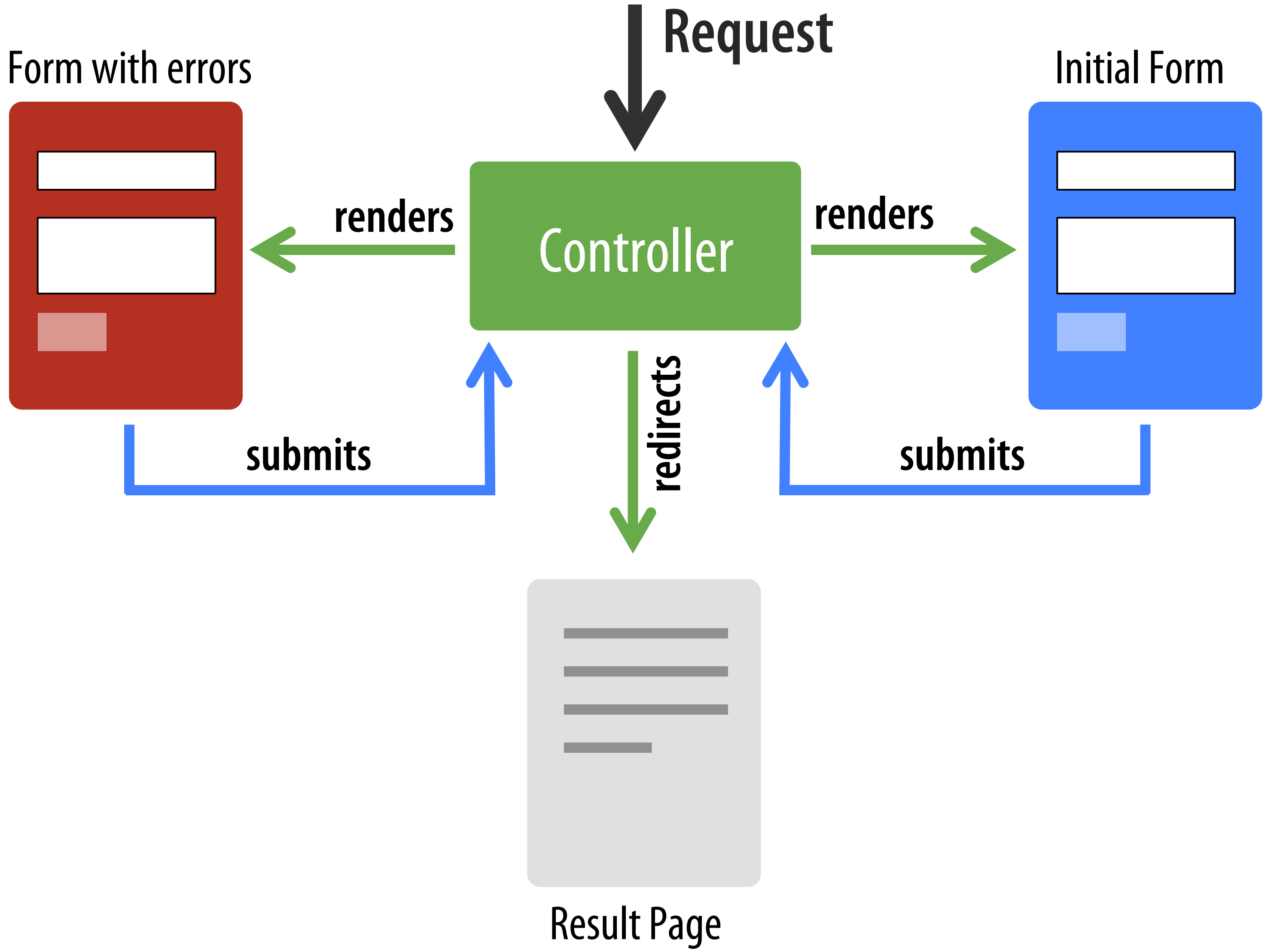
It's recommended to use keys as the content of the original messages, to make translations easier to maintain.

Translating validation messages (2 of 2)

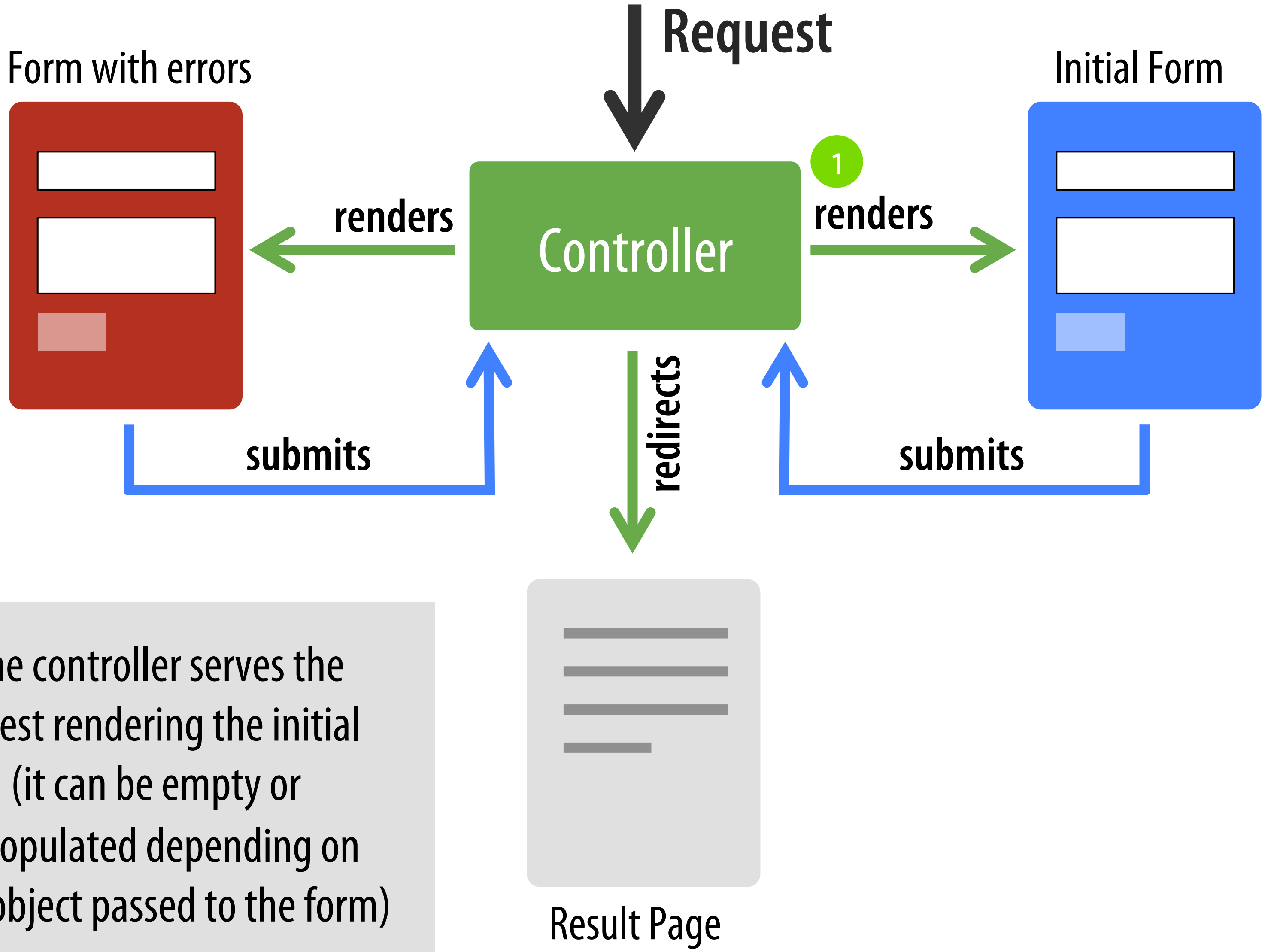
```
<!-- app/Resources/translations/validators.en.xlf -->
<?xml version="1.0"?>
<xliff version="1.2"
xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext"
original="file.ext">
    <body>
      <trans-unit id="author.name.not_blank">
        <source>author.name.not_blank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

Processing forms

The Big Picture of handling Symfony forms

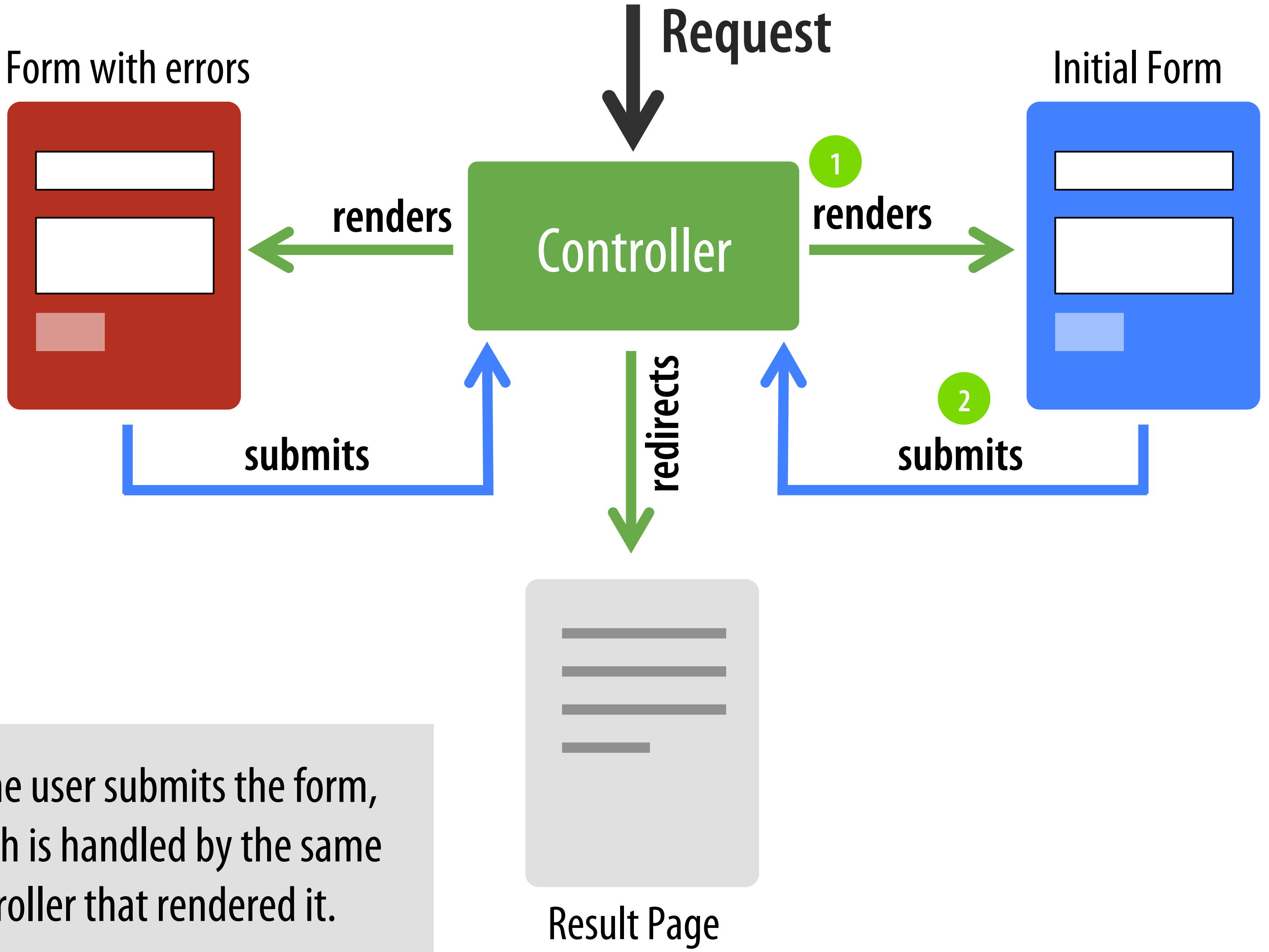


The Big Picture of handling Symfony forms



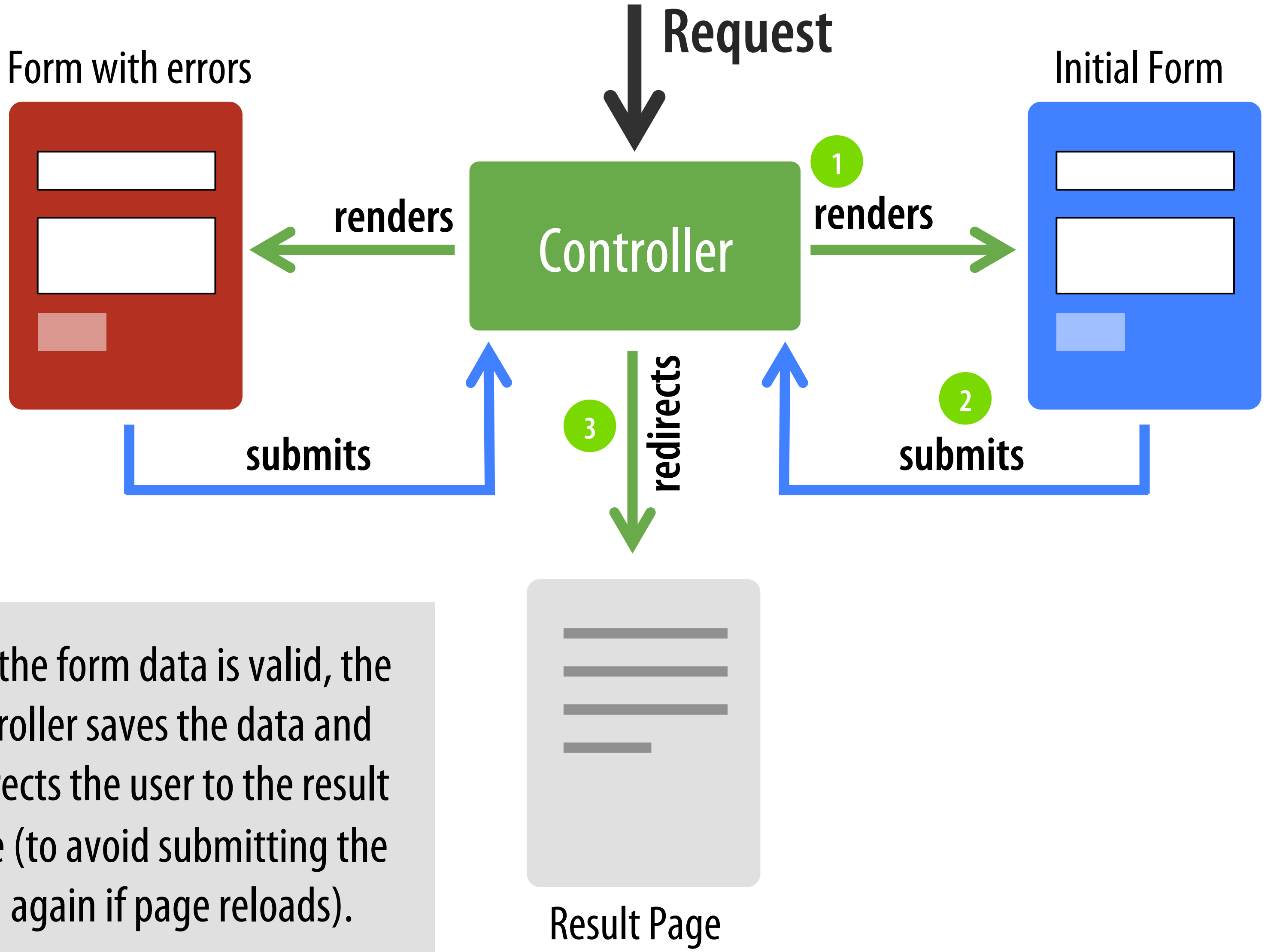
1. The controller serves the request rendering the initial form (it can be empty or prepopulated depending on the object passed to the form)

The Big Picture of handling Symfony forms



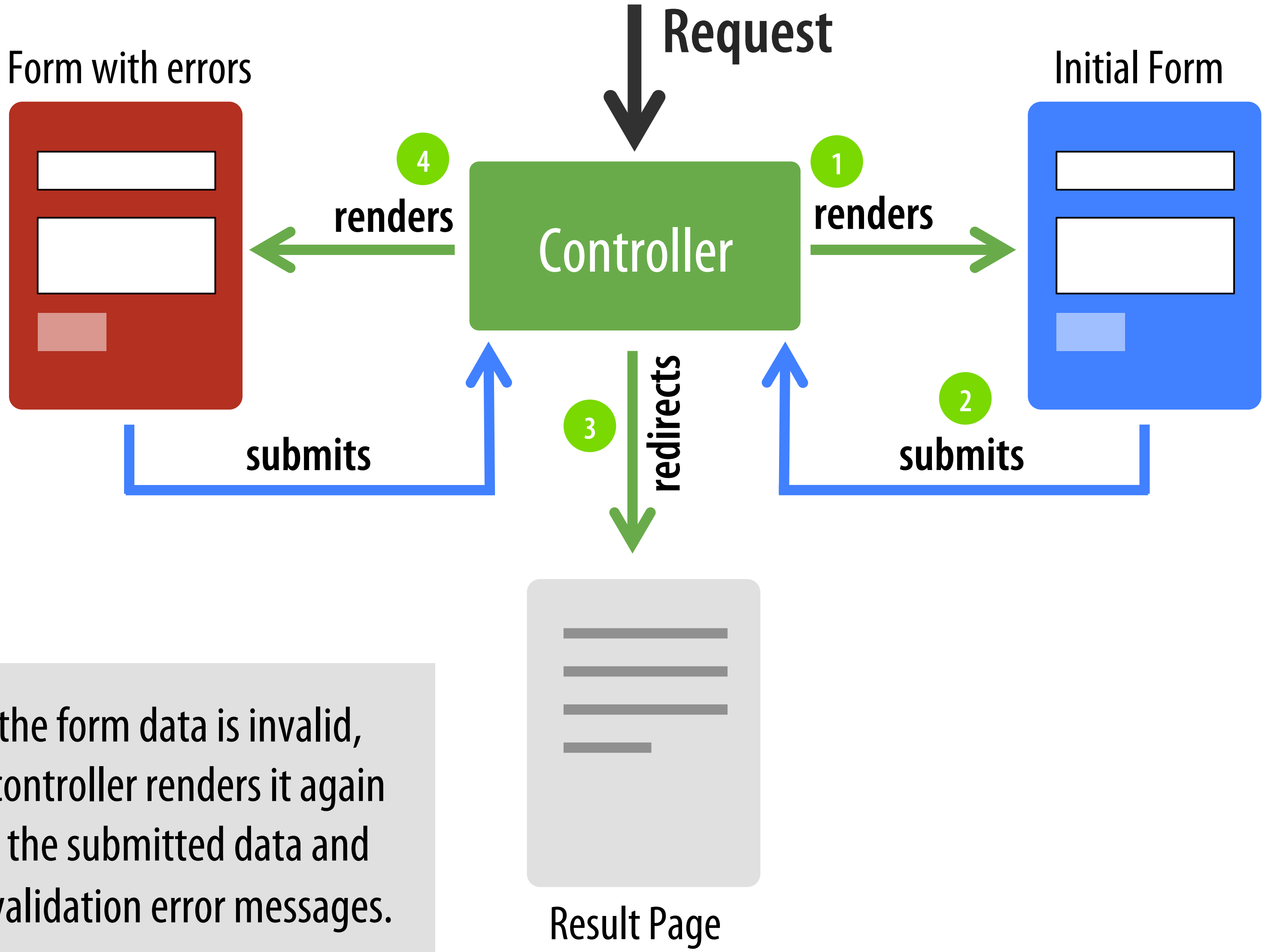
2. The user submits the form, which is handled by the same controller that rendered it.

The Big Picture of handling Symfony forms



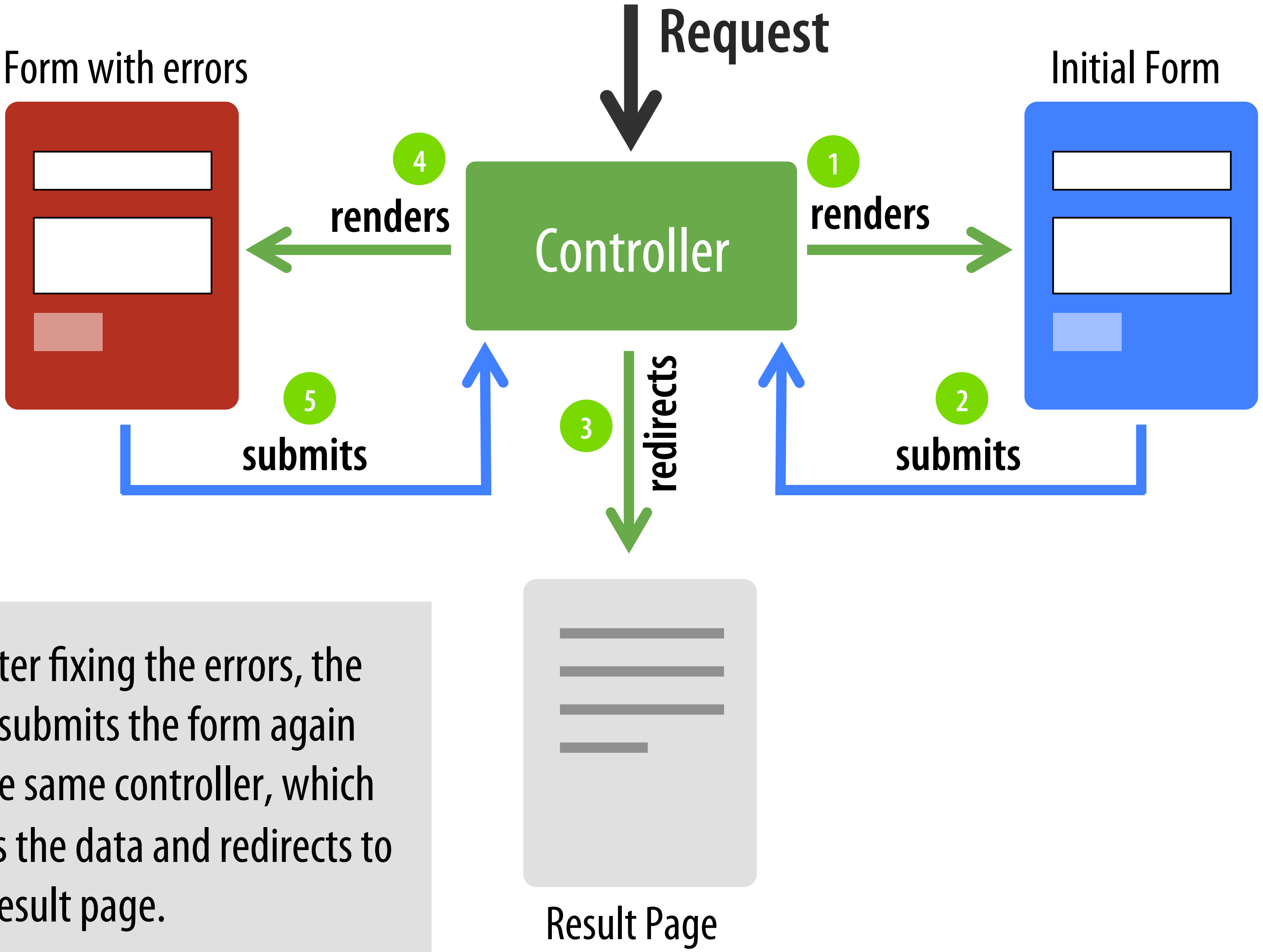
3. If the form data is valid, the controller saves the data and redirects the user to the result page (to avoid submitting the form again if page reloads).

The Big Picture of handling Symfony forms



4. If the form data is invalid, the controller renders it again with the submitted data and the validation error messages.

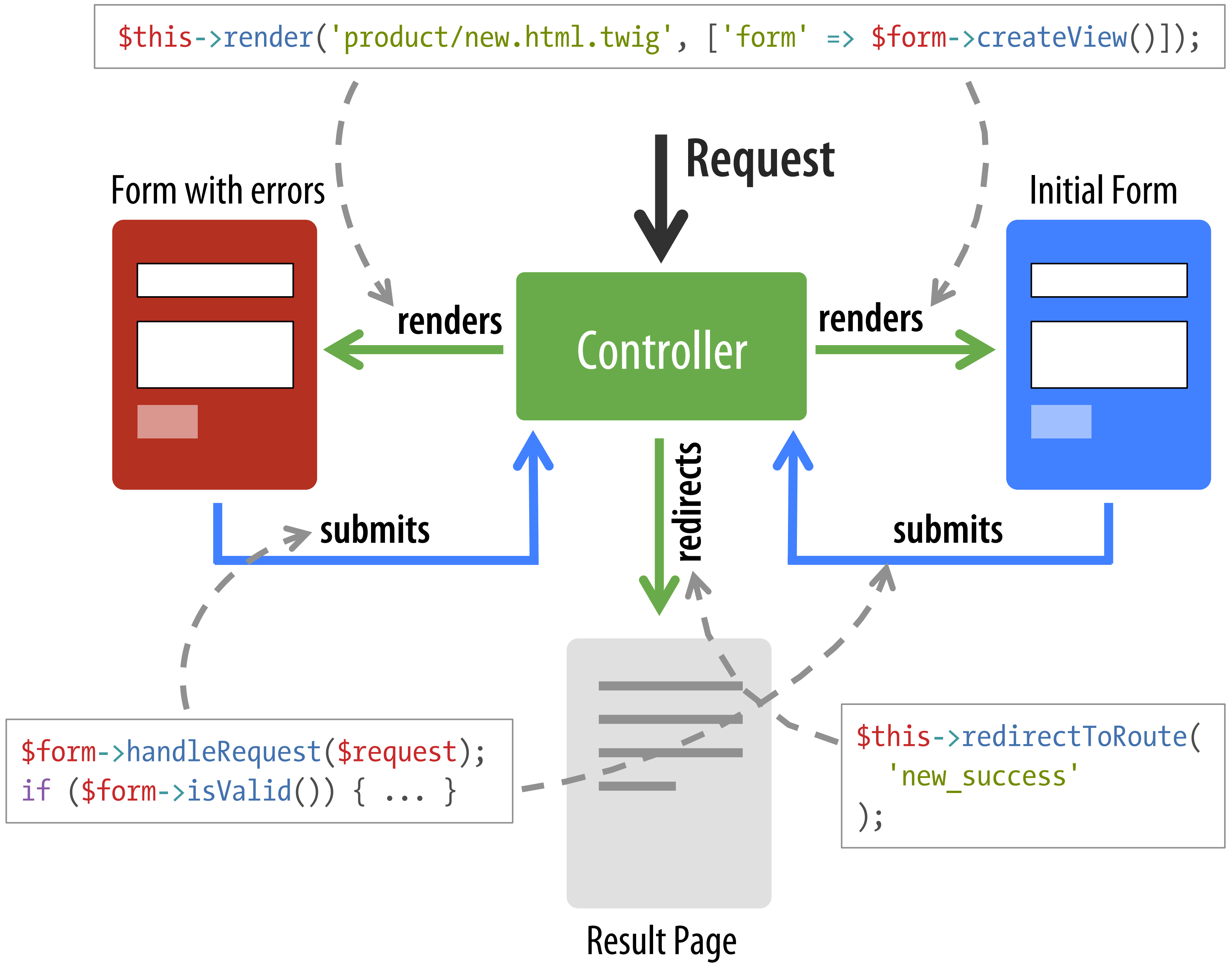
The Big Picture of handling Symfony forms



5. After fixing the errors, the user submits the form again to the same controller, which saves the data and redirects to the result page.

Handling and processing forms in practice

```php
use AppBundle\Entity\Product;
use AppBundle\Form\ProductType;

public function newAction(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', array(
        'form' => $form->createView()
    ));
}
```



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

```
$form->handleRequest($request);
if ($form->isValid()) { ... }
```

```
$this->redirectToRoute(
  'new_success'
);
```

Rendering forms

Fast form rendering for prototypes

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

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) }}
    {{ form_errors(form) }}


    {{ form_row(form.name) }}
    {{ form_row(form.price) }}
{{ form_end(form) }}
```

Advanced form rendering

```
{{ form_start(form) }}
```

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

```
{{ form_end(form) }}
```

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) }}
```

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.name) }}
```

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

Configuring the form behavior

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

{{ form_start(form, {
     '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) }}
   {{ form_errors(form) }}

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

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

Detailed form rendering

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

It renders the label for the given form field.

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

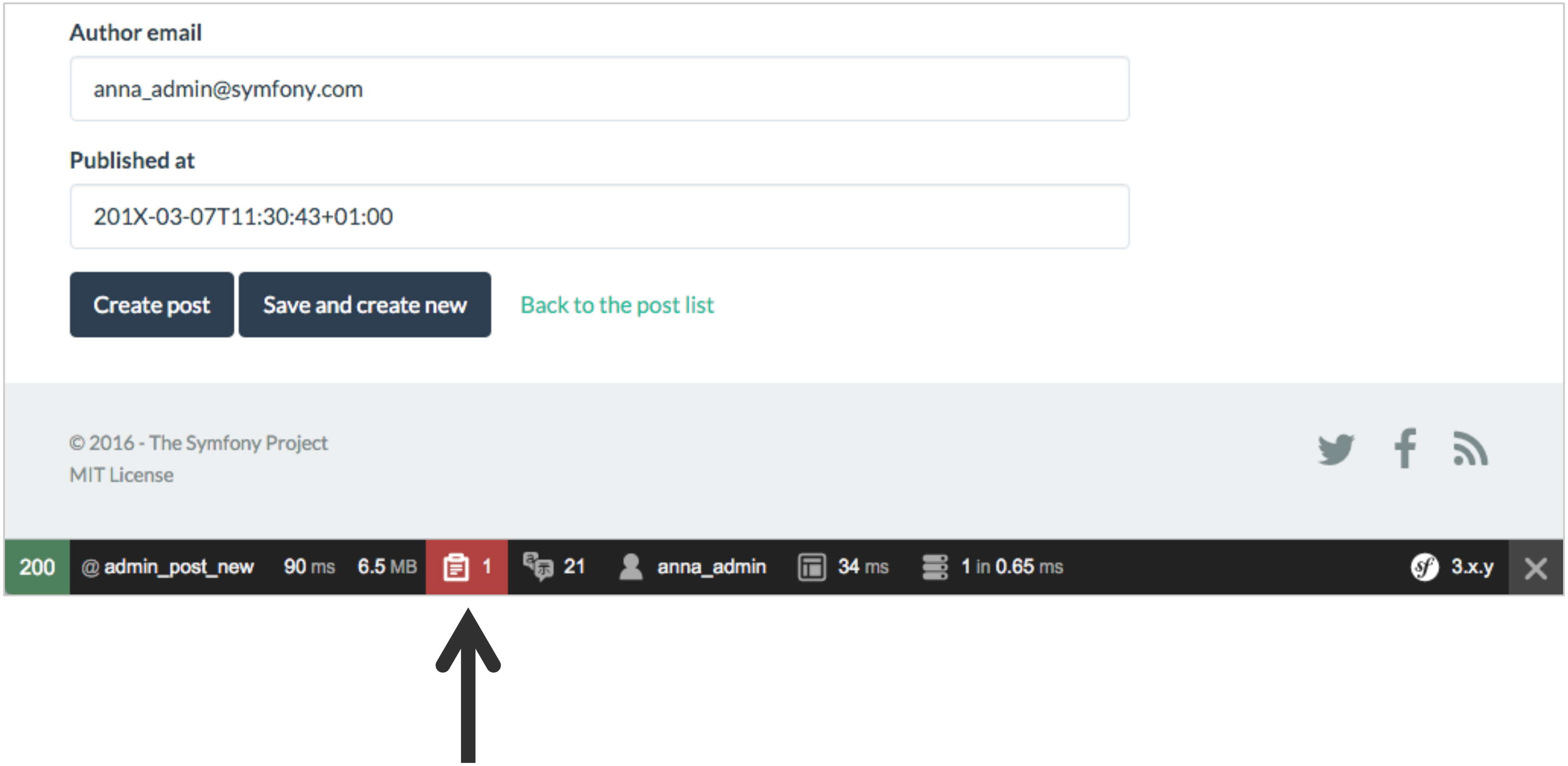
It renders the errors specific to the given form field (if any).

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

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

Form debugging

Form errors in the web debug toolbar



Author email

anna_admin@symfony.com

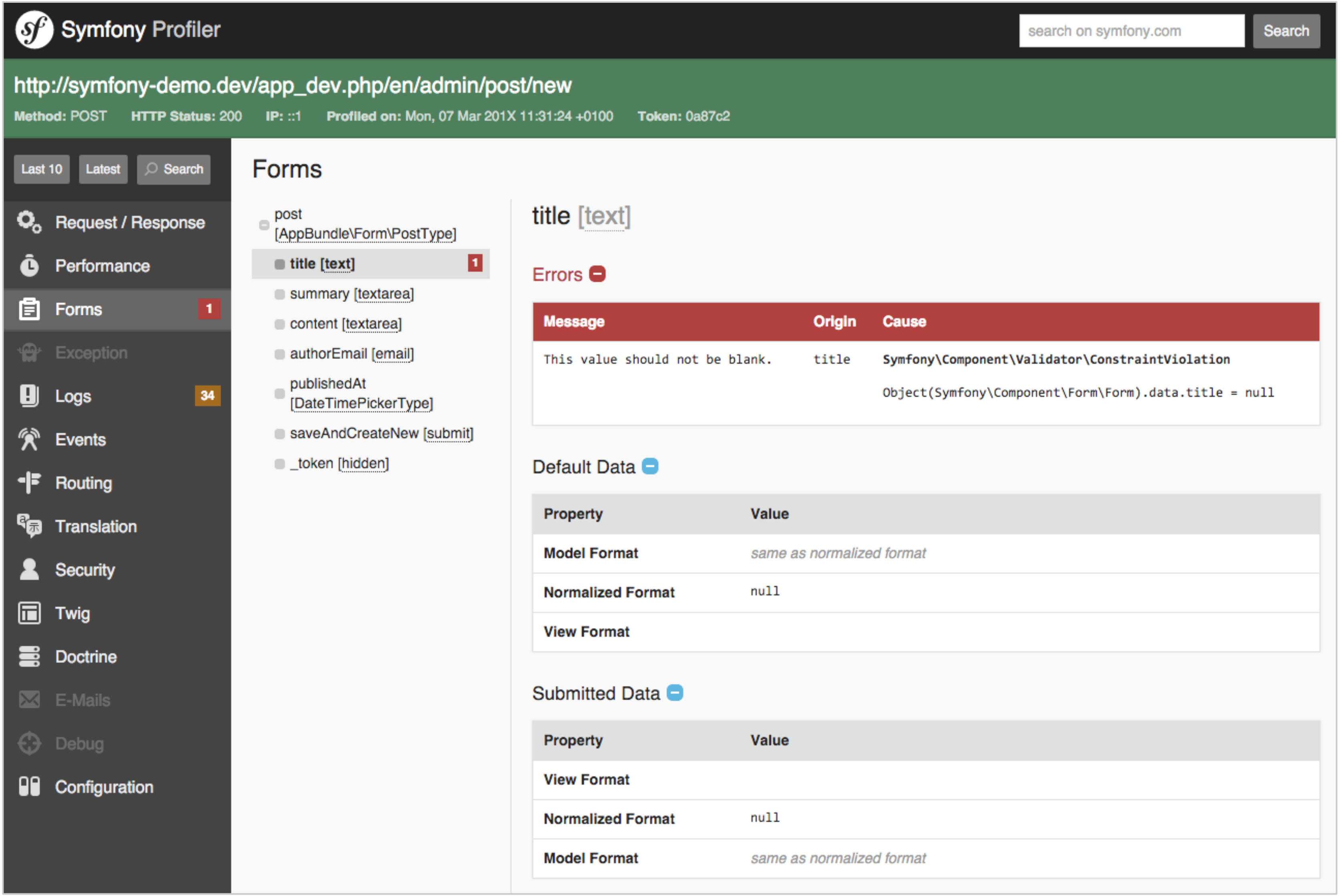
Published at

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

© 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
```

Form errors in the Symfony profiler



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

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

SensioLabs

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

SensioLabs Deutschland
Neusser Str. 27-29
50670 Köln
Germany
Tel: +49 221 66 99 27 50
contact@sensiolabs.de

We also provide worldwide **on-site services**. Contact us at: sensiolabs.com/en/contact

Our products and services

SensioLabs Insight

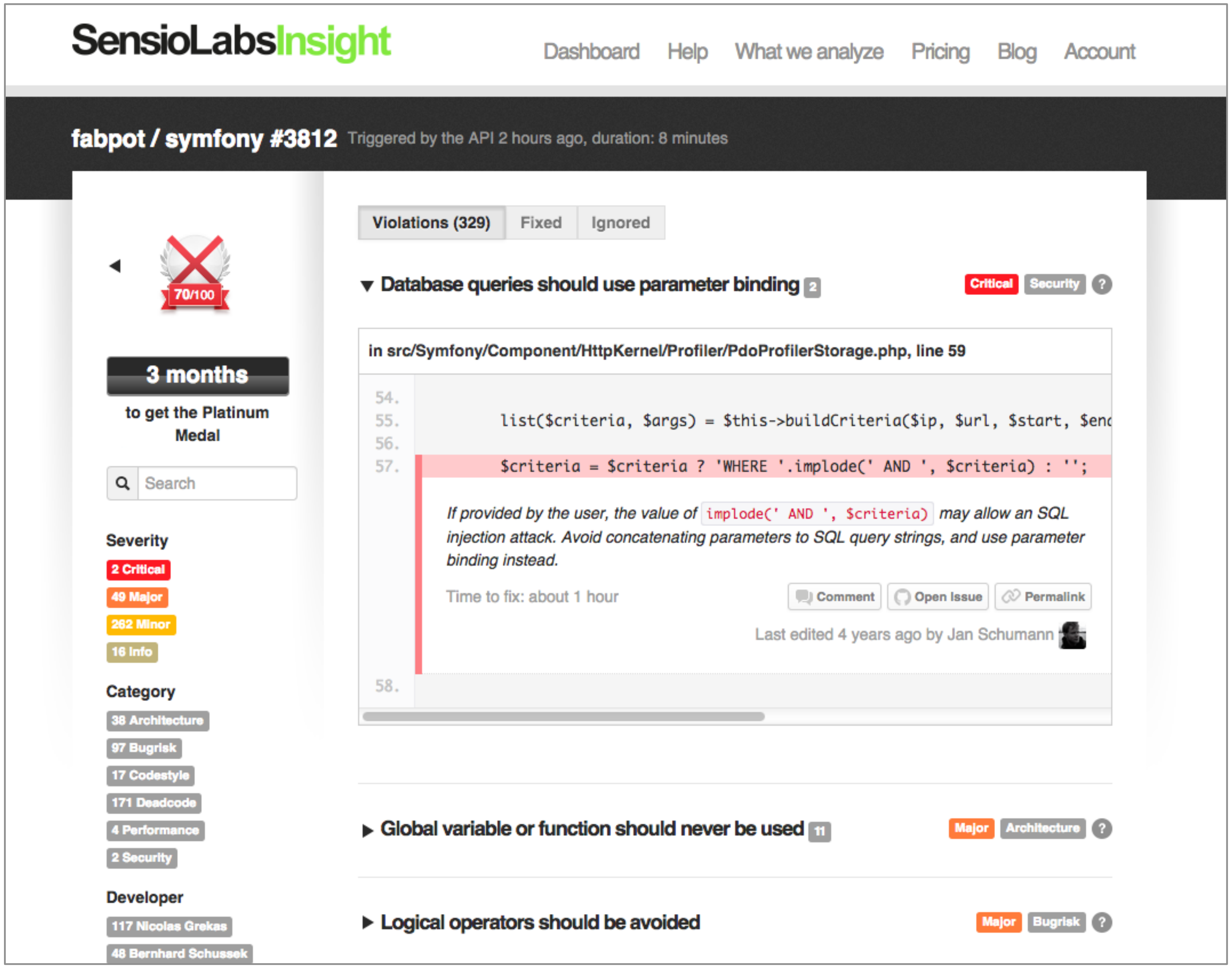
Blackfire

Consulting

Training

SensioLabs Insight

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

70/100

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

Violations (329) | Fixed | Ignored

▼ **Database queries should use parameter binding** 2 — Critical, Security

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

```
        list($criteria, $args) = $this->buildCriteria($ip, $url, $start, $enc

        $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

▶ **Global variable or function should never be used** 11 — Major, Architecture

▶ **Logical operators should be avoided** — Major, Bugrisk

It analyzes the **quality** of your **code** continuously.

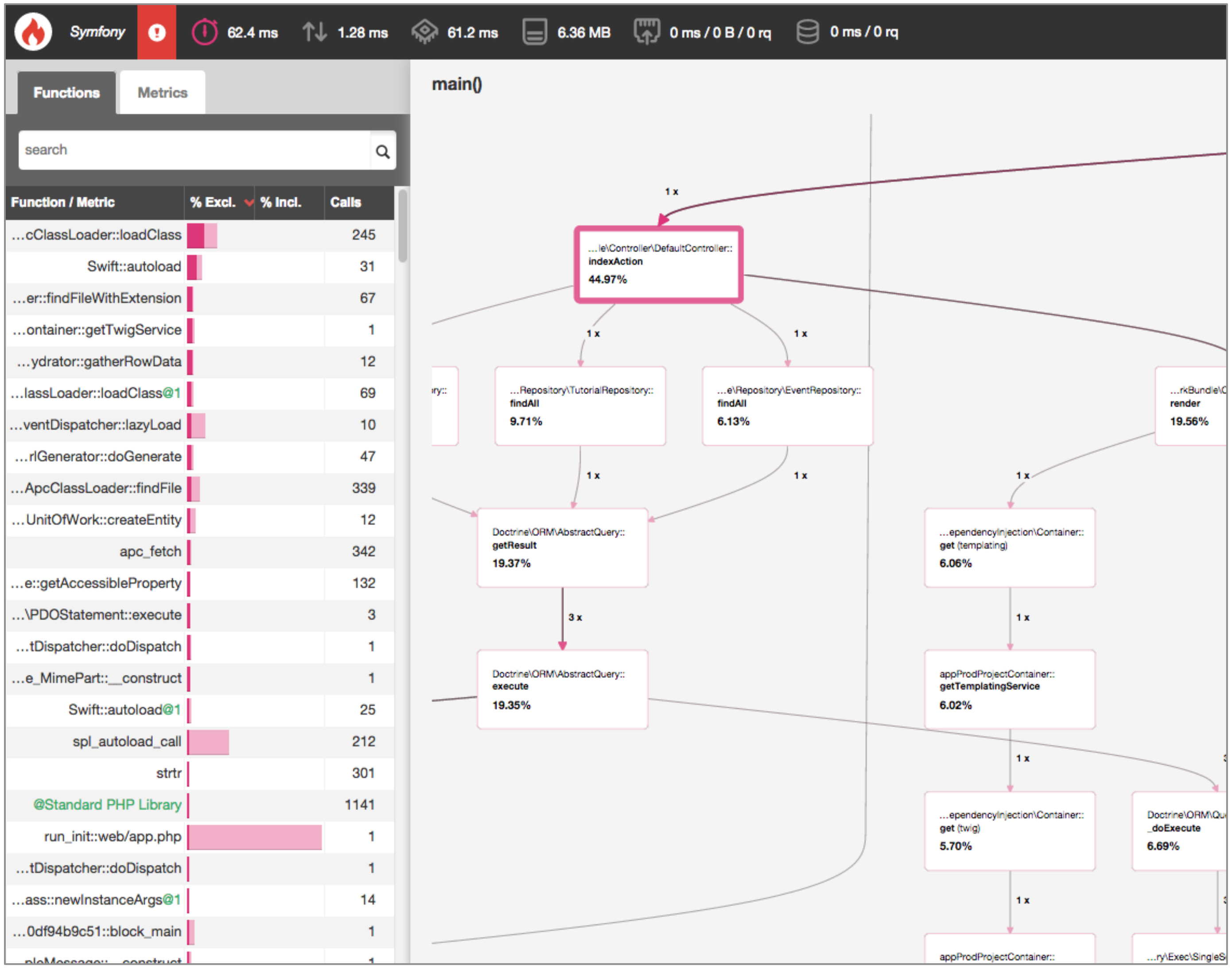
All problems are displayed on context and they provide **detailed solutions**.

It works for **any PHP application**, not only Symfony.

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

Blackfire

blackfire.io


Symfony
62.4 ms
1.28 ms
61.2 ms
6.36 MB
0 ms / 0 B / 0 rq
0 ms / 0 rq
Functions
Metrics
search
main()
...le\Controller\DefaultController:: indexAction 44.97%
...Repository\TutorialRepository:: findAll 9.71%
...e\Repository\EventRepository:: findAll 6.13%
...rkBundle\C render 19.56%
Doctrine\ORM\AbstractQuery:: getResult 19.37%
Doctrine\ORM\AbstractQuery:: execute 19.35%
...ependencyInjection\Container:: get (templating) 6.06%
appProdProjectContainer:: getTemplatingService 6.02%
...ependencyInjection\Container:: get (twig) 5.70%
Doctrine\ORM\Qu _doExecute 6.69%
appProdProjectContainer::
...ry\Exec\SingleS
1 x
3 x

Function / Metric	% Excl.	% Incl.	Calls
...cClassLoader::loadClass			245
Swift::autoload			31
...er::findFileWithExtension			67
...ontainer::getTwigService			1
...ydrator::gatherRowData			12
...lassLoader::loadClass@1			69
...ventDispatcher::lazyLoad			10
...rlGenerator::doGenerate			47
...ApcClassLoader::findFile			339
...UnitOfWork::createEntity			12
apc_fetch			342
...e::getAccessibleProperty			132
...\PDOStatement::execute			3
...tDispatcher::doDispatch			1
...e_MimePart::__construct			1
Swift::autoload@1			25
spl_autoload_call			212
strtr			301
@Standard PHP Library			1141
run_init::web/app.php			1
...tDispatcher::doDispatch			1
...ass::newInstanceArgs@1			14
...0df94b9c51::block_main			1
...pleMessage::__construct			1

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

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

SensioLabs Training Department

Address

Balthasarstraße 79

50670 Köln

Germany

Phone

+49 221 16 53 54 0

Email

trainings@sensiolabs.de