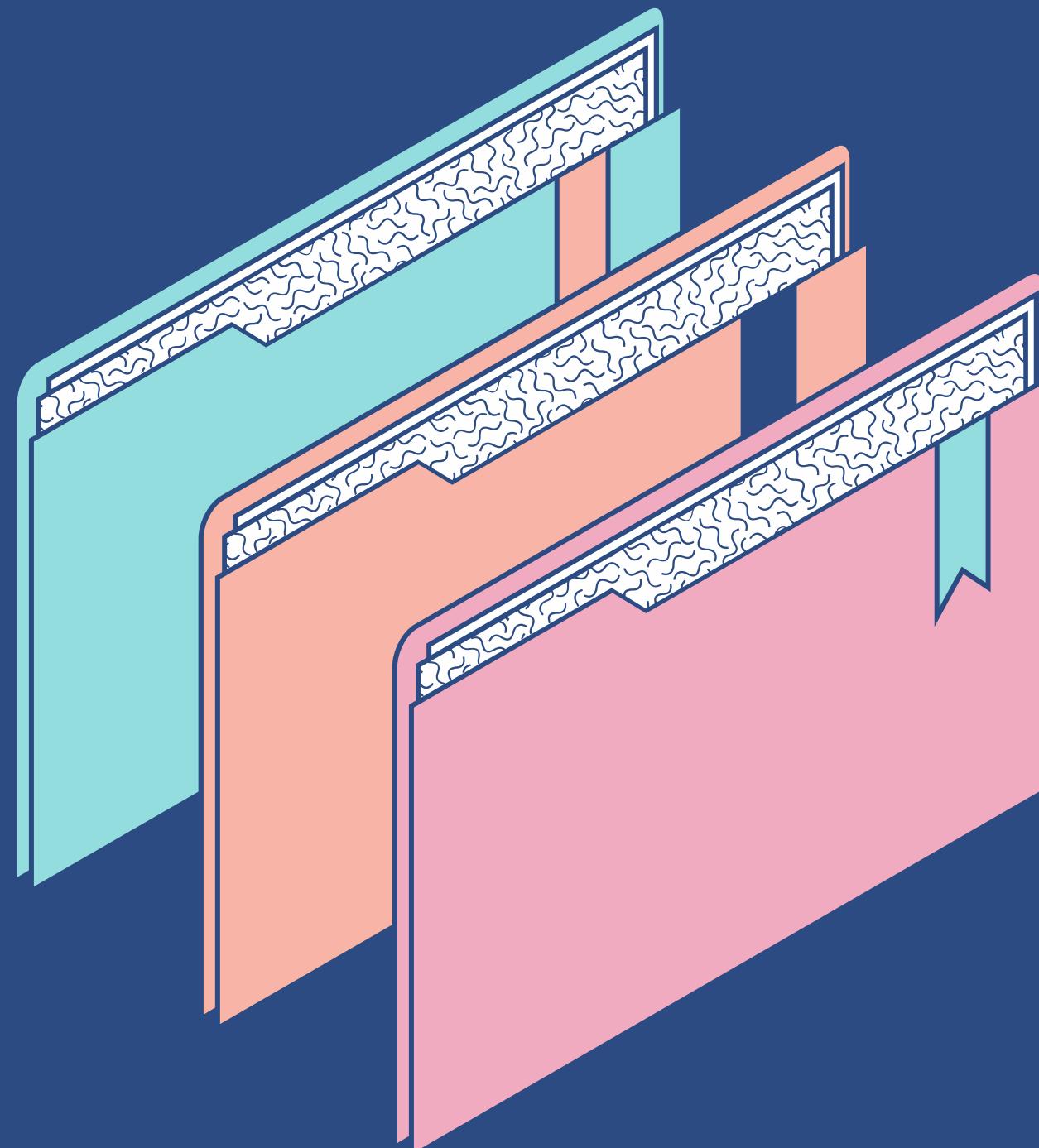




Symfony

SOMMAIRE



1. Présentation du Framework
2. Installation et prérequis
3. Structure des projets
4. Première page Symfony
5. Routes
6. Controller
7. Vues TWIG

Présentation de Symfony

3



Symfony

Présentation de Symfony

Symfony : ensemble de composants PHP, Framework d'application Web, philosophie et communauté.

Un Framework permet aux développeurs de gagner du temps en réutilisant des modules génériques pour se concentrer sur d'autres domaines. Il va nous aider à développer mieux et plus vite !

Un Framework vous apporte la certitude que l'on développe une application parfaitement conforme aux règles métier, structurée, maintenable et évolutive.

Présentation de Symfony

Symfony fournit :

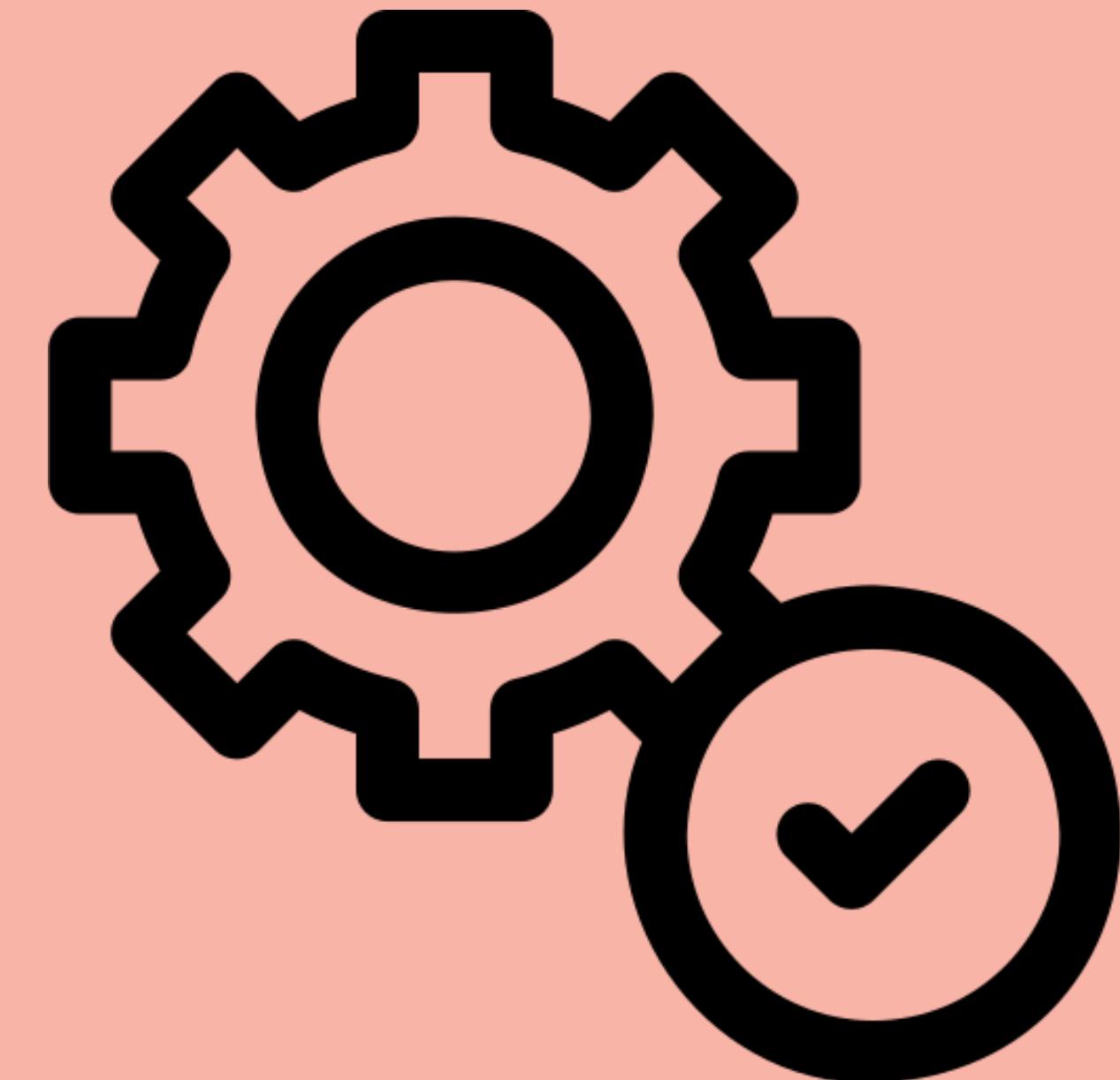
- Une méthodologie :
 - Convention d'écriture et d'organisation
 - Discipline du code produit
 - Architecture MVC
- Des outils :
 - CRUD
 - Génération administration
 - Plugins

Présentation de Symfony

Most Popular Backend Frameworks



Installation et prérequis



Prérequis

Architecture :

- Windows (> windows10)
- Linux
- MacOs

Gestionnaire de dépendances :

- Scoop
- Composer

Serveur Symfony :

- Symfony-cli

Scoop - Présentation

Scoop est un gestionnaire de packages, permettant d'automatiser l'installation et la maintenance d'outils système et de packages.

Cet outil n'est pas obligatoirement, mais permettra de faciliter par la suite certaines opérations nécessaires au bon fonctionnement de Symfony.

Plus d'infos : <https://scoop.sh/>



Scoop - Installation

Depuis une interface POWERSHELL :

```
iex (new-object net.webclient).downloadstring('https://get.scoop.sh')
```



Symfony-cli - Présentation

Symfony-cli est un logiciel développé et maintenu par Symfony.

Il permet :

- La création d'applications Symfony à partir des squelettes applicatifs
- Un serveur local pour exécuter votre projet sur votre poste
- Un outil pour vérifier les problèmes de sécurité

Symfony-cli - Installation

Depuis une interface POWERSHELL :

```
scoop install symfony-cli
```

NB : Avec symfony, nous n'avons plus besoin de serveur local !



Composer - Présentation

Composer est un gestionnaire de dépendances écrit en PHP.

Il permet d'automatiser l'installation et la gestion des dépendances d'un projet très facilement.

Cet outil est facultatif également, mais permettra de faciliter par la suite certaines opérations nécessaires au bon fonctionnement de Symfony.

Composer - Installation

Le package **Composer-setup.exe** est téléchargeable à l'adresse suivante :

<https://getcomposer.org/download/>

Création d'un projet

Création d'une Webapp à l'aide de composer :

```
composer create-project symfony/website-skeleton mon_super_site
```

NB : Il existe de nombreuses façon de créer un projet, cette méthode permet de générer une structure complète

Utilisation du serveur symfony-cli

Ouvrez un nouveau terminal au sein d'un projet Symfony.

Démarrage du serveur : [symfony server:start](#)

Par défaut, le serveur sera alors accessible à l'adresse suivante : <http://127.0.0.1:8000>

Arrêt du serveur : [symfony server:stop](#)

Structure d'un projet

17



Structure des projets

```
> bin  
> config  
> migrations  
> public  
> src  
> templates  
> tests  
> translations  
> var  
> vendor  
  .env  
  .env.test  
  .gitignore  
  composer.json  
  composer.lock  
  phpunit.xml.dist  
  symfony.lock
```

Structure des projets

- Dossier **bin** : 2 fichiers
 - Console (exécutable pour manager le projet)
 - Phpunit (exécutable pour tester l'application)
- Dossier **config** : Contient toutes les informations de configuration du projet
- Dossier **migrations** : Contient le SQL généré du projet
- Dossier **public** : Contient les fichiers et dossiers accessibles au public
- Dossier **templates** : Contient les Templates HTML des pages
- Dossier **test** : Contient les tests du projet
- Dossier **translations** : Contient les différentes traductions (langues) du projet
- Dossier **var** : Contient les variables et fichiers temporaires

Structure des projets

- Dossier **vendor** : Contient les dépendances du projet
- Fichier **.env** : Contient les variables d'environnement
- Fichier **.env.test** : Contient les variables de tests
- Fichier **.gitignore** : Fichier git contenant les fichiers et dossiers à exclure
- Fichier **composer.json** : Fichier contenant les dépendances du projet
- Fichier **composer.lock** : Fichier contenant les dépendances du projet
- Fichier **phpunit.xml.dist** : Fichier de configuration des tests du projet
- Fichier **symfony.lock** : Fichier de configuration des dépendances installées sur le projet

Première page Symfony

21



You're seeing this page because you haven't configured any homepage URL.



Welcome to
Symfony 5.0.8



/mnt/c/Users/csrsc/blog/

Your application is now ready and you can start working on it.

Première page Symfony

Pour créer une page dans Symfony, il faut, au minimum :

- Une **route** : pour faire le lien entre une URL et une méthode d'un contrôleur
- Un **contrôleur** : qui contient des méthodes, chacune, en générale, associée à une route
- Une **méthode** : permet l'execution d'une action précise, généralement en lien avec une route

Premier Controller

Symfony permet d'automatiser la mise en place de Controller à l'aide de la commande :

```
symfony console make:controller NomController
```

où *NomController* correspond au nom (*CamelCase*)

Premier Controller

```
<?php  
  
namespace App\Controller;  
  
use Symfony\Component\HttpFoundation\Response;  
  
class LuckyController  
{  
    public function number(): Response  
{  
        $number = random_int(0, 100);  
  
        return new Response(  
            'Lucky number: '.$number  
        );  
    }  
}
```

Créez un nouveau Controller "LuckyController"

Dupliquez y le code ci-joint



Premier Controller

Ce Controller est composé d'une méthode qui calcule un nombre aléatoire et retourne la réponse.

Ce code n'utilise pas les vues de Symfony, et n'est pour le moment pas fonctionnel (il n'est pas possible de l'appeler) car aucune route n'est paramétrée.

Première route

```
<?php  
  
namespace App\Controller;  
  
use Symfony\Component\HttpFoundation\Response;  
use Symfony\Component\Routing\Annotation\Route;  
  
class LuckyController  
{  
    #[Route("/lucky/number", name:"app_lucky_number")]  
    public function number(): Response  
    {  
        $number = random_int(0, 100);  
  
        return new Response(  
            'Lucky number: '.$number  
        );  
    }  
}
```

Une route permet de faire le lien entre une URL et une méthode de Controller.

Modifiez le Controller créé précédemment pour y ajouter une route

Première vue

Pour pouvoir afficher notre résultat au sein d'une page HTML valide, et au format désiré, nous avons besoin d'afficher notre résultat au sein d'une vue.

Par défaut, Symfony utilise la syntaxe TWIG, permettant de simplifier l'écriture de nos vues.

Pour fonctionner, il est nécessaire d'installer twig au sein du projet :

```
composer require twig
```

Première vue

```
<?php

namespace App\Controller;

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

class LuckyController extends AbstractController
{
    #[Route("/lucky/number", name:"app_lucky_number")]
    public function number()
    {
        $number = random_int(0, 100);

        return $this->render(
            'lucky/number.html.twig',
            [
                'number' => $number
            ]
        );
    }
}
```

Modifiez ensuite votre Controller pour que
celui-ci fasse appel à votre vue

Première vue



A screenshot of a file explorer showing a project structure. The 'templates' folder contains 'conference' and 'lucky' subfolders. 'lucky' contains 'index.html.twig' and 'number.html.twig'. Other files like 'base.html.twig' and 'nom.html.twig' are also visible.

```

> bin
< config
> packages
> routes
  bundles.php
  preload.php
  routes.yaml
  services.yaml
> migrations
> public
< src
  Controller
    .gitignore
    LuckyController.php
  Entity
  Repository
    Kernel.php
< templates
  conference
  lucky
    index.html.twig
    number.html.twig
  nom
  base.html.twig

```

```

1  {# templates/lucky/number.html.twig #}
2  <h1>Your lucky number is {{ number }}</h1>

```

Enfin, créez un nouveau répertoire "lucky" au sein de vos templates

Ajoutez y un fichier number.html.twig

Ajoutez y le code ci-joint.

Vous pouvez ensuite tester le bon fonctionnement en lançant votre serveur

Routing avec Symfony



Présentation

Une route permet de diriger une URL vers une méthode de Controller, appelée Action.

Une route peut être définie selon les formations de fichiers :
XML, Classe PHP, Annotations, Attributs (php>8)

La méthode Attributs étant la plus récente et la plus commode à utiliser, c'est cette méthode que nous étudierons par la suite

Attributes

Pour pouvoir utiliser les routes Attributes, il est nécessaire d'ajouter :

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

au sein de votre Controller.

Structure d'une route

Une route peut être constante : `/blog` ou dynamique : `/blog/{slug}`

Dans le cas d'une route dynamique, `{slug}` correspond à une variable qu'il sera possible de récupérer au sein du Controller.

Exemples : `/blog/42` `/blog/lorem-ipsum` `/blog/titi-32/tata`

Structure d'une route

Une route est composée au minimum d'un chemin (path) et d'un nom (name)

```
# [Route('/blog/{slug}', name:'article_blog')]  
public function article($slug)  
{  
    ...  
}
```

Structure d'une route

Ces variables peuvent être mises par défaut grâce à "defaults"

Ces variables peuvent être soumises à validation de format regex via "requirements"

```
# [Route('/{id}', name:'index', requirements:['id'=>'\d+'], defaults:['id'=>1])]  
public function index($id)  
{  
    ...  
}
```

NB : Il est également possible de définir le mode de récupération (CRUD) à l'aide du paramètre "methods"

Structure d'une route

Il est possible de cumuler plusieurs variables d'URL en séparant chaque paramètre de la route :

```
# [Route('/page/{page}/{subpage}', name:'blog_index')]  
public function indexAction($page, $subpage)  
{  
    echo $page.' '.$subpage;  
}
```

Génération d'URL - Partie controller

Symfony nous permet de générer dynamiquement une URL à partir du nom de la route. Cette méthode permet d'inclure facilement des variables au sein de la route, et de l'identifier par nom et non par valeur.

En cas de modification de la valeur de la route, tous vos liens resteront ainsi fonctionnels.

```
$this->generateUrl('nom_de_la_route', [$variables]);
```

```
$url = $this->generateUrl('produit_details', ['id' => $idProduit]);
```



Génération d'URL - Partie vues

Cette logique est également disponible au sein des vues TWIG

```
 {{ url('nom_route', {'page': 'toto', 'vars2': 'titi'}) }}
```

```
<a href="{{ url('mon_profil', { 'id': userId }) }}>Mon profil</a>
```

Controlleurs

39



Présentation

Un Controller Symfony se nomme toujours en **CamelCase**, avec le suffixe **Controller**



LuckyController.php

Les méthodes du controller se nomment toujours en **lowerCamelCase**

```
public function randomNumber()
```

Response

Une action renvoie toujours une Response

Il existe différents types de Response : JsonReponse, RedirectResponse, HttpResponse ...

La plus utilisée est la réponse générique Response, qu'il est nécessaire d'importer :

```
use Symfony\Component\HttpFoundation\Response;
```

On peut ensuite retourner notre return en tant que réponse :

```
public function index(){
    |
    return new Response('Ma response');
}
```

Response

Dans l'état, notre réponse n'est pas du HTML, mais un code de réponse.

En héritant de la classe `AbstractController`, il est possible d'utiliser la méthode `render()`.

Cette méthode permettra aux Actions de récupérer une vue, et d'afficher le contenu de la vue compilée avec les différentes variables envoyées.

Response

```
#Route("/", name:"page")
public function index($variable)
{
    // votre code

    return $this->render('default/index.html.twig',
        ['variable' => $variable]
    );
}
```



Exercice 1

- Créez un nouveau projet Symfony : `exerciceSymfony`
- Créez 2 nouveaux Controllers : `dateController` & `colorController`
- Créez une route et sa méthode associée permettant de retourner en tant que Response (sans TWIG) l'heure actuelle (H/min/s) sur la route suivante : `localhost/time/now`
- Créez une nouvelle route et sa méthode associée permettant d'afficher "Bleu" ou "Vert" à l'écran au sein d'une page HTML valide (TWIG) sur les routes suivantes :
`localhost/color/blue` | `localhost/color/green`

Manipuler les requests

L'objet **Request** contient toutes les données envoyées par l'utilisateur **ET** par son navigateur.

En passant l'objet **Request** en paramètre, on peut alors aisément le manipuler.

Manipuler les requests

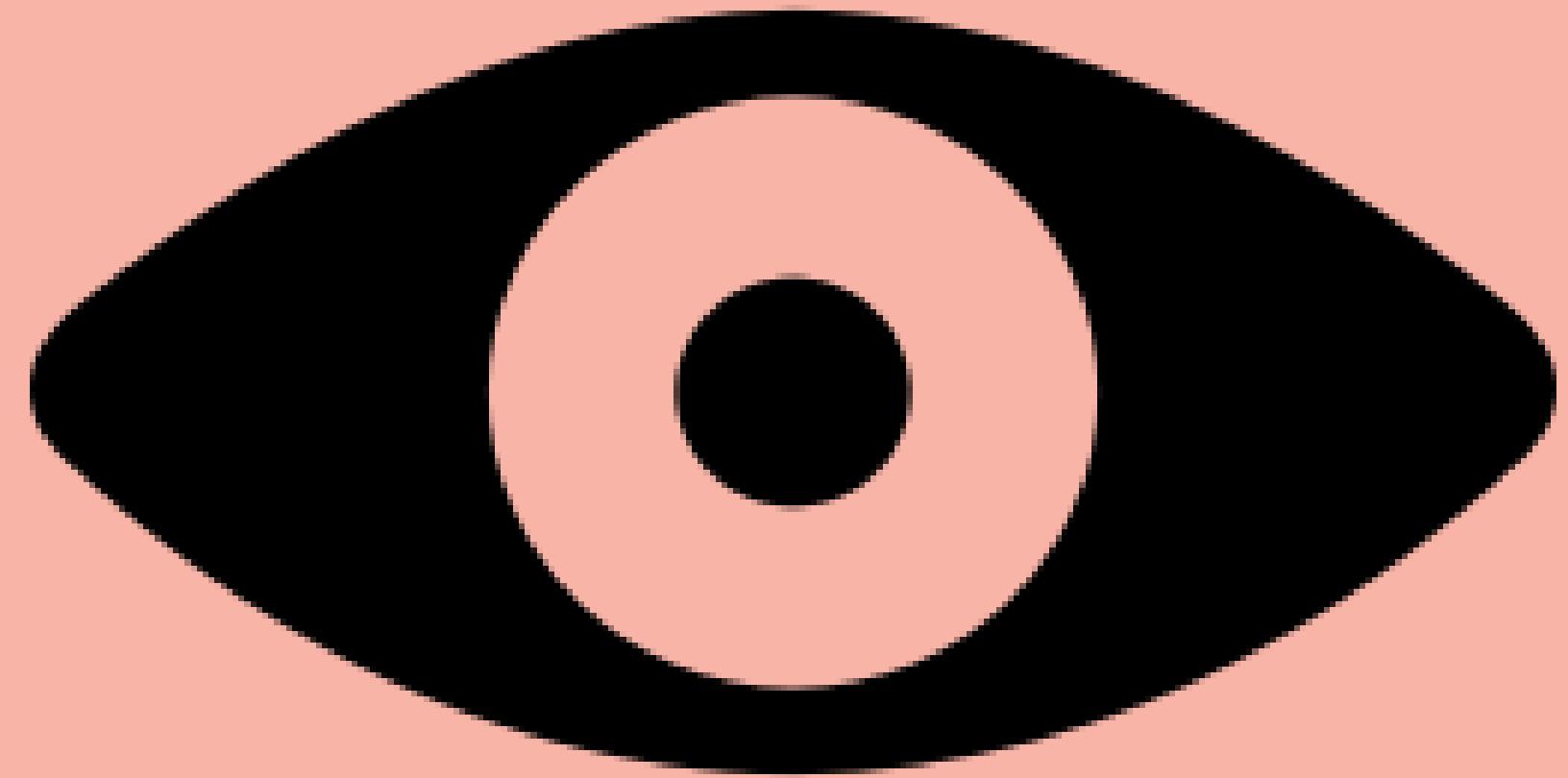
```
use Symfony\Component\HttpFoundation\Request;
class ColorController extends AbstractController
{
    public function req(Request $request)
    {
        $request->isXmlHttpRequest(); // is it an Ajax request?
        $request->getPreferredLanguage(array('en', 'fr'));
        // retrieves GET and POST variables respectively
        $request->query->get('page');
        $request->request->get('page');
        // retrieves SERVER variables
        $request->server->get('HTTP_HOST');
        // retrieves an instance of UploadedFile identified by foo
        $request->files->get('foo');
        // retrieves a COOKIE value
        $request->cookies->get('PHPSESSID');
        // retrieves an HTTP request header, with normalized, lowercase keys
        $request->headers->get('host');
        $request->headers->get('content_type');
    }
}
```

Exercice 2

Reprenez le projet `exerciceSymfony`

- Ajoutez une méthode (route, méthode et vue) permettant de déterminer l'IP du client.
Vous pourrez utiliser la méthode `getClientIp()` de l'objet `request`.
- Ajoutez un menu permettant de rediriger vers les pages ip, times et dates, accessible depuis ces pages

Vues TWIG



Présentation

Un Template (View) est le meilleur moyen d'organiser et restituer le code HTML.

Les vues dans Symfony sont écrites en TWIG, un moteur de modèle flexible, rapide et sécurisé.

TWIG offre les mêmes fonctionnalités que PHP, nous l'utiliserons uniquement à des fins de simplification de l'affichage front

Présentation

```
<body>
    <h1><?php echo $page_title ?></h1>
    <ul id="navigation">
        <?php foreach ($navigation as $item): ?>
        <li>
            <a href="<?php echo $item->getHref() ?>">
                <?php echo $item->getCaption() ?>
            </a>
        </li>
    <?php endforeach ?>
    </ul>
</body>
```

```
<body>
    <h1>{{ page_title }}</h1>
    <ul id="navigation">
        {% for item in navigation %}
            <li><a href="{{ item.href }}>{{ item.caption }}</a></li>
        {% endfor %}
    </ul>
</body>
```

Affichage

La syntaxe Twig est basée sur uniquement trois constructions:

- {{ ... }}, utilisé pour afficher le contenu d'une variable ou le résultat de l'évaluation d'une expression;
- {%- %}, utilisé pour exécuter une logique, telle qu'une condition ou une boucle;
- {# ... #}, utilisé pour ajouter des commentaires au modèle

(contrairement aux commentaires HTML, ces commentaires ne sont pas inclus dans la page rendue)

Variables

TWIG est très puissant lorsqu'il s'agit de manipuler des données.
En TWIG, on accède aux données d'un tableau de la même façon qu'aux données d'un objet :

(php tab) \$tab["param1"] => {{ tab.param1 }} (twig)

(php obj) \$tab.param1 => {{ tab.param1 }} (twig)

Tests

Il est possible d'écrire des tests, la syntaxe est très proche de celle de PHP.
Attention toutefois, les opérateurs de comparaison ne s'écrit pas de manière identique.
Le && de PHP s'écrit "and" dans TWIG, et le || de PHP s'écrit "or" dans TWIG.

```
{% if condition1 or condition2 %}  
    <p>Instruction 1</p>  
    {% elseif condition3 and condition4 %}  
        <p>Instruction 2</p>  
    {% else %}  
        <p>Instruction 3</p>  
    {% endif %}
```

Boucles

Il n'existe que la boucle for dans TWIG (elle est un peu l'équivalent d'un foreach).

```
{% for i in 1..10 %}

{% endfor %}
```

La boucle ci-dessous est une boucle qui parcours une "collection" users (l'équivalent du foreach).

```
<ul>
    {% for user in users %}
        <li>{{ user.username }}</li>
    {% endfor %}
</ul>
```

Attention la syntaxe est inversée par rapport au PHP

Boucles

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.indexo	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindexo	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

Héritage

TWIG permet l'héritage via un `extends` dans les templates enfants

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

Dans les templates parents, on définit des "blocks" que l'on surcharge dans les enfants

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

```
{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>
```



Exercice 3

- Reprenez le projet `exerciceSymfony`
- Déplacez votre menu, de sorte à le rendre accessible sur toutes les pages en l'incluant uniquement au sein de la page parent
- Pour la couleur verte, affichez un message "Tout va bien!".
- Pour la couleur bleue, affichez un message "Risque potentiel"

Filtres

Il est possible d'appliquer des filtres sur les variables pour effectuer des transformations (majuscules, dates, minuscules ...)

L'utilisation des filtres se fait à l'aide d'un pipe " | " à la suite de la variable à afficher.
(Attention, l'ordre des filtres est parfois important !)

```
 {{ variable|upper }}
```

Liste des filtres TWIG : <https://twig.symfony.com/doc/3.x/>

Assets

TWIG permet de gérer facilement les liens vers vos ressources (images, js, php ...)

Pour cela, il sera nécessaire d'inclure le bundle d'assets (à l'aide de PowerShell) :

```
composer require symfony/asset
```

Cette fonctionnalité permet à TWIG de créer dynamiquement le lien vers la ressource à laquelle vous souhaitez accéder

Assets

Par défaut, l'ensemble de vos assets (ressources) doivent se trouver au sein du répertoire public de Symfony, et sera accessible de la façon suivante :

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



Exercice 4

- Ajoutez une image pour chaque page de votre projet (date, ip, color) illustrant le contenu de la page
- Ajoutez un fichier css pour chaque page de votre projet afin d'en améliorer le design