

PROGRAMMATION Web Dynamique Framework Symfony

Zarrouk ELYES

zarrouk.elyes@gmail.com



Plan





Vue d'ensemble de Symfony

Qu'est-ce qu'un framework ?

- Un framework est une boîte à outils conçue par plusieurs développeurs à destination d'autres.
- L'objectif premier d'un framework est d'améliorer la productivité des développeurs qui l'utilisent.
- Contrairement aux CMS(**Content Management System**), un framework est destiné à des développeurs, et non à des novices en informatique.
- L'apprentissage d'un framework est un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !



Vue d'ensemble de Symfony

Qu'est-ce qu'un framework ?

Avantages

- Un framework offre des briques prêtes à être utilisées
- Un code bien organisé est un code facilement maintenable et évolutif
- un code qui respecte les standards de programmation
- un code source maintenu par des développeurs attitrés

Inconvénients

- La courbe d'apprentissage est très élevée
- Connaître certaines bonnes pratiques telles que l'architecture MVC est obligatoire.
- il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !



Vue d'ensemble de Symfony

Qu'est-ce que Symfony?

- Symfony est un framework PHP très populaire, français, et très utilisé dans le milieu des entreprises.
- Créé par SensioLabs, agence Web française
- La première version de Symfony est sortie en 2005, il y a plus de 10 ans
- Symfony2 est sortie en août 2011.
- Les versions 3 et 4 restent sur la même base de code que la version 2
- Symfony adopte une solide politique de rétrocompatibilité (*Backward Compatibility*) entre chaque version. Cela signifie que si vous choisissez Symfony pour votre projet, vous êtes assuré de pouvoir suivre les mises à jour du framework, et donc de bénéficier des nouvelles fonctionnalités et résolutions de bogues, sans réécrire complètement votre application à chaque fois.



Vue d'ensemble de Symfony

Comment installer Symfony?

Avant de créer votre première application Symfony, vous devez:

- Installer PHP7 et ses extensions Ctype, iconv, JSON, PCRE, Session, SimpleXML et Tokenizer.
- Installer Composer qui sera utilisé pour installer des paquets PHP
- <https://getcomposer.org/download/>
- Installer Symfony qui crée dans votre ordinateur un binaire appelé symfony qui fournit tous les outils dont vous avez besoin pour développer votre application localement.
- <https://symfony.com/download>

Vue d'ensemble de Symfony

Créer une application Symfony à travers la console

❑ Pour créer une application web traditionnelle

> symfony new my_project_name --full (--version=4.4)

Ou

- composer create-project symfony/website-skeleton my_project_name
- composer create-project symfony/framework-standard-edition my_project_name (y inclut symfony3)

❑ Pour lancer son application web

>cd my-project/

- symfony server:start

Ou si vous voulez changer l'adresse IP et le port

- php bin/console server:dump <IP:PORT>
- Php -S 127.0.0.1:<PORT>

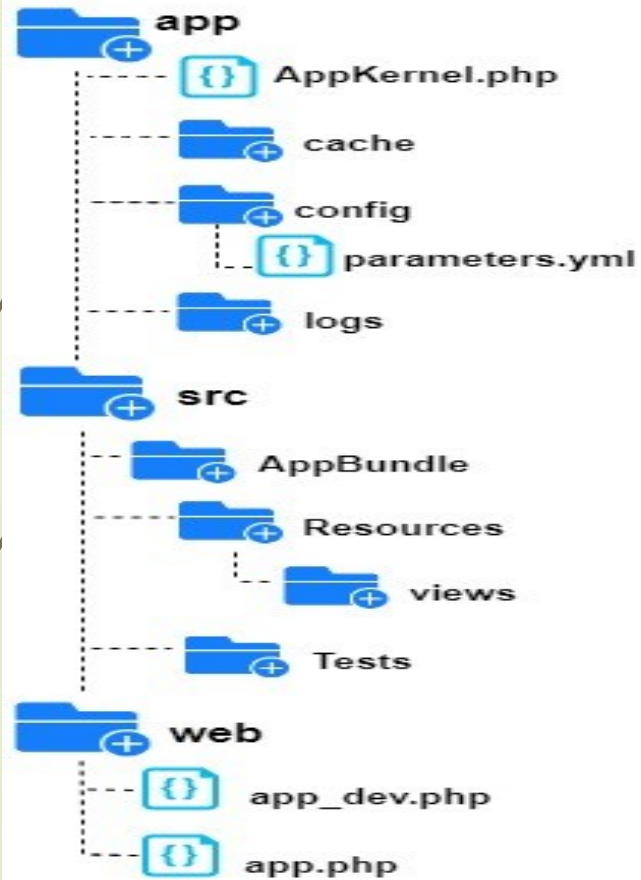
Vue d'ensemble de Symfony

Un projet Symfony est composé de :

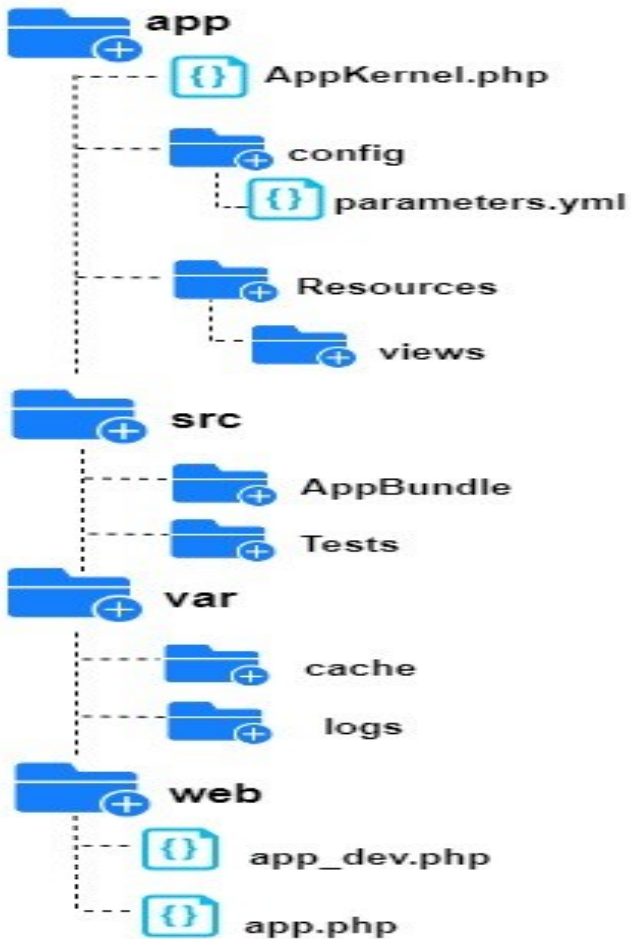
- ❑ **/Bin** : Fichiers binaires des bibliothèques utilisées par l'application Web
- ❑ **/Config** : C'est ici que nous configurerons Symfony lui-même, mais aussi les plugins (ou bundles) que nous installerons par la suite.
- ❑ **/public** : Ce répertoire contient tous les fichiers destinés à vos visiteurs : images, fichiers CSS et JavaScript, etc. Il contient également le contrôleur frontal (index.php).
- ❑ **/src** : le répertoire dans lequel on mettra le code source !
- ❑ **/var** : Ce répertoire contient tout ce que Symfony va écrire durant son exécution : les logs, le cache, et d'autres fichiers nécessaires à son bon fonctionnement.
- ❑ **/vendor** : Ce répertoire contient toutes les bibliothèques externes à notre application.
- ❑ **/template** : Ce répertoire contiendra tous les templates de notre application

Vue d'ensemble de Symfony

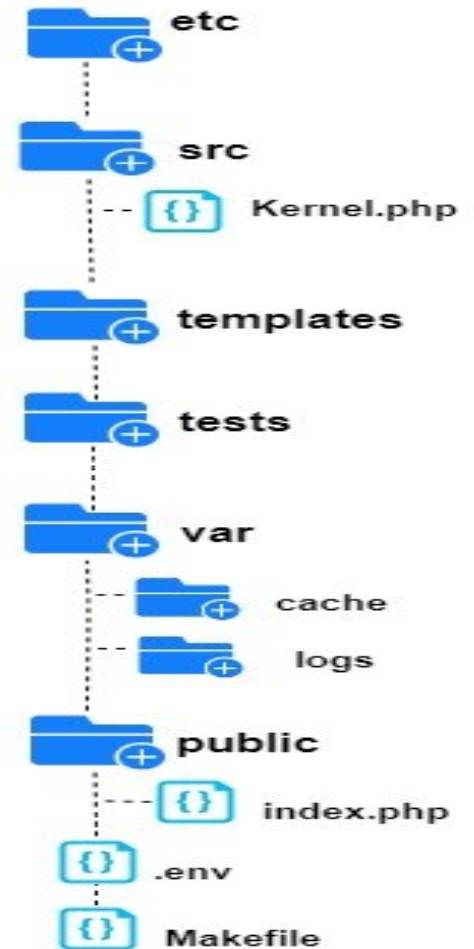
Symfony 2



Symfony 3



Symfony 4



Vue d'ensemble de Symfony

Le contrôleur frontal

- ❑ Le contrôleur frontal (front controller) est le point d'entrée de votre application. C'est le fichier par lequel passent toutes vos pages. Dans Symfony, le contrôleur frontal se situe dans le répertoire/public, il s'appelle également `index.php` .
- ❑ le but du contrôleur frontal n'est pas de faire quelque chose, mais d'être un point d'entrée de notre application. Il se limite donc (en gros) à appeler le noyau (kernel) de Symfony en disant :
« On vient de recevoir une requête, transforme-la en réponse s'il-te-plaît. »
- ❑ Symfony offre deux environnements de travail:
 - ✓ Environnement de production
 - ✓ Environnement de développement (par défaut)

Vue d'ensemble de Symfony

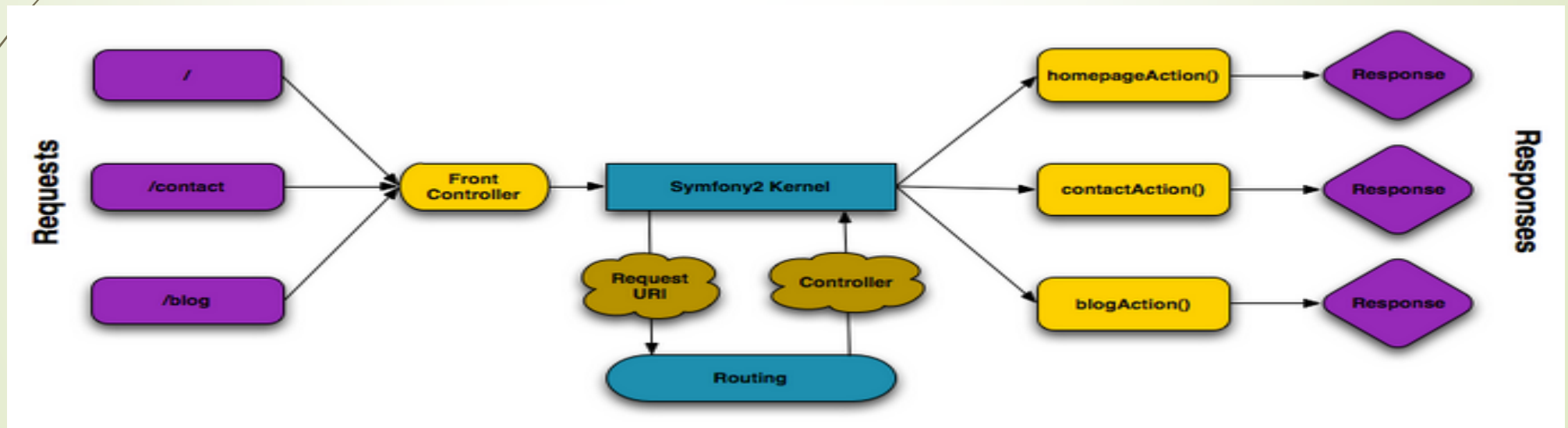
L'Architecture conceptuelle

- ❑ Symfony respecte bien entendu cette architecture **MVC**
- ❑ MVC signifie « Modèle / Vue / Contrôleur ». C'est un découpage très répandu pour développer les sites Internet, car il sépare les couches selon leur logique propre :
 - ✓ **Le contrôleur** : Son rôle est d'analyser et traiter la requête de l'utilisateur et de générer sa réponse en agissant avec les modèles et les vues.
 - ✓ **Le modèle** : Son rôle est de gérer vos données et votre contenu. Dans le cas typique d'une base de données, le modèle offre des méthodes pour mettre à jour ces données (insertion suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ses données...
 - ✓ **La vue** : Sa première tâche est d'afficher les données qu'elle a récupérées auprès du modèle. Sa seconde tâche est de recevoir tous les actions de l'utilisateur (clic de souris, sélection d'une entrées, boutons, ...). Ses différents événements sont envoyés au contrôleur.

Vue d'ensemble de Symfony

Flux applicatif d'une requête

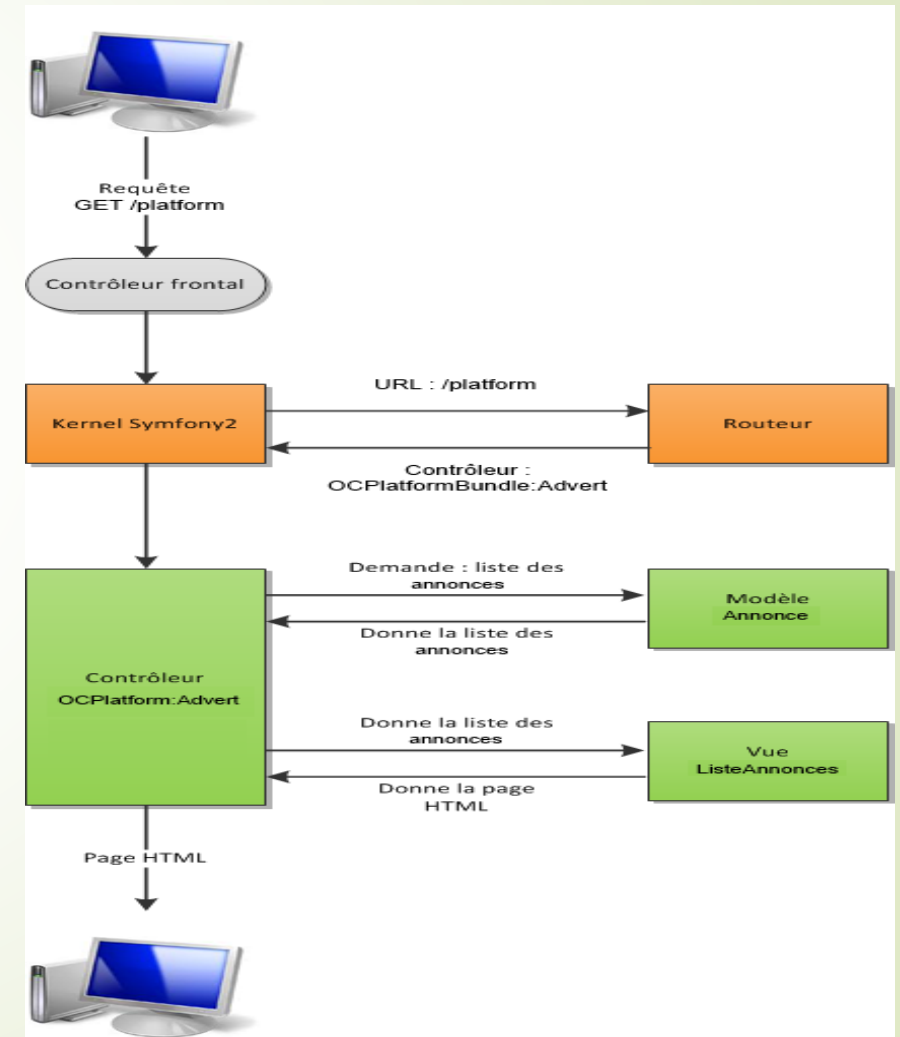
- ❑ Toutes les requêtes sont interceptées par le contrôleur frontal
- ❑ Le noyau trouve le contrôleur via le système de routage et exécute l'action adéquate
- ❑ Une réponse est produite



Vue d'ensemble de Symfony

Flux applicatif d'une requête

1. Le visiteur demande la page /platform ;
2. Le contrôleur frontal reçoit la requête, charge le Kernel et la lui transmet ;
3. Le Kernel demande au Routeur quel contrôleur exécuter pour l'URL /platform. Ce Routeur est un composant Symfony qui fait la correspondance entre URL et contrôleurs. Le Routeur fait donc son travail, et dit au Kernel qu'il faut exécuter le contrôleurOCPlatform:Advert ;
4. Le Kernel exécute donc ce contrôleur. Le contrôleur demande au modèleAnnonce la liste des annonces, puis la donne à la vueListeAnnonces pour qu'elle construise la page HTML et la lui retourne. Une fois cela fini, le contrôleur envoie au visiteur la page HTML complète.



Les bases de symfony

Le routage

- Faire correspondre une URL et une action d'un contrôleur
- Par exemple, nous pourrions avoir une route qui dit « Lorsque l'URL appelée est /hello-world, alors le contrôleur à exécuter est »
- Le contrôleur à exécuter est un paramètre obligatoire de la route, mais il peut bien sûr y en avoir d'autres.
- Le rôle du routeur est donc de trouver la bonne route qui correspond à l'URL appelée, et de retourner les paramètres de cette route.

```
# config/routes.yaml
```

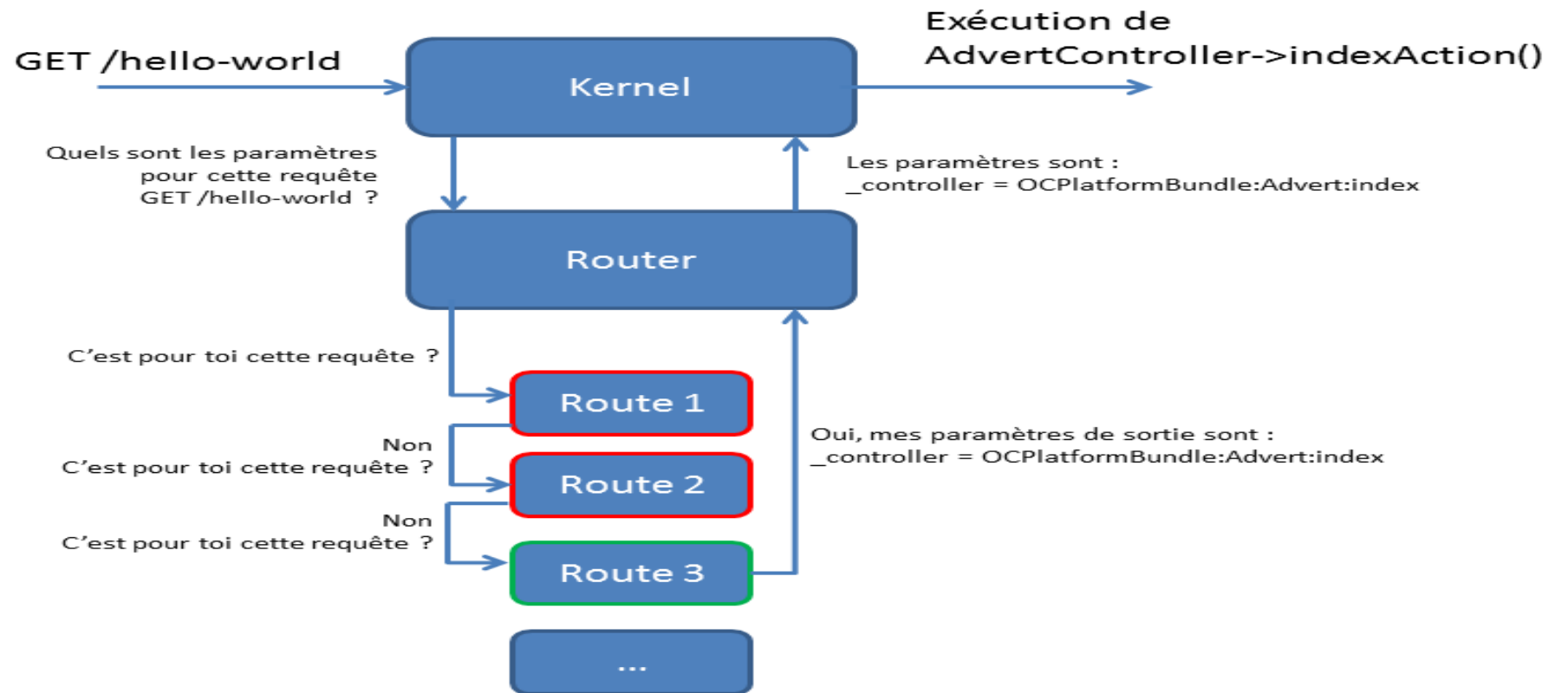
```
hello_the_world:
```

```
  path:    /hello-world
```

```
  controller: App\Controller\AdvertController::index
```


Les bases de symfony

Le routage



Les bases de symfony

Le routage

On peut définir également les routes dans symfony à travers les annotations directement depuis les contrôleurs.

- Il nous faut installer la recette `annotations`, qui permet d'utiliser plusieurs types d'annotations, dont celle qui nous intéresse pour définir les routes. Exécutez donc **composer require annotations**

```
oc_advert_index:  
  path:      /advert  
  controller: App\Controller\AdvertController::index
```

```
/**  
 * @Route("/advert", name="oc_advert_index")  
 */
```

Les bases de symfony

Le routage

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Twig\Environment;
class AdvertController
{
    /**
     * @Route("/advert", name="oc_advert_index")
     */
    public function index(Environment $twig)
    {
        $content = $twig->render('Advert/index.html.twig', ['name' => 'elyes']);

        return new Response($content);
    }
}
```

Les bases de symfony

Le routage avec paramètres

```
<?php
// src/Controller/AdvertController.php

namespace App\Controller;

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

class AdvertController
{
    // ...

    /**
     * @Route("/advert/view/{id}", name="oc_advert_view")
     */
    public function view($id)
    {
        // $id vaut 5 si l'URL appelée est /advert/view/5

        return new Response("Affichage de l'annonce d'id : ".$id);
    }
}
```

Les bases de symfony

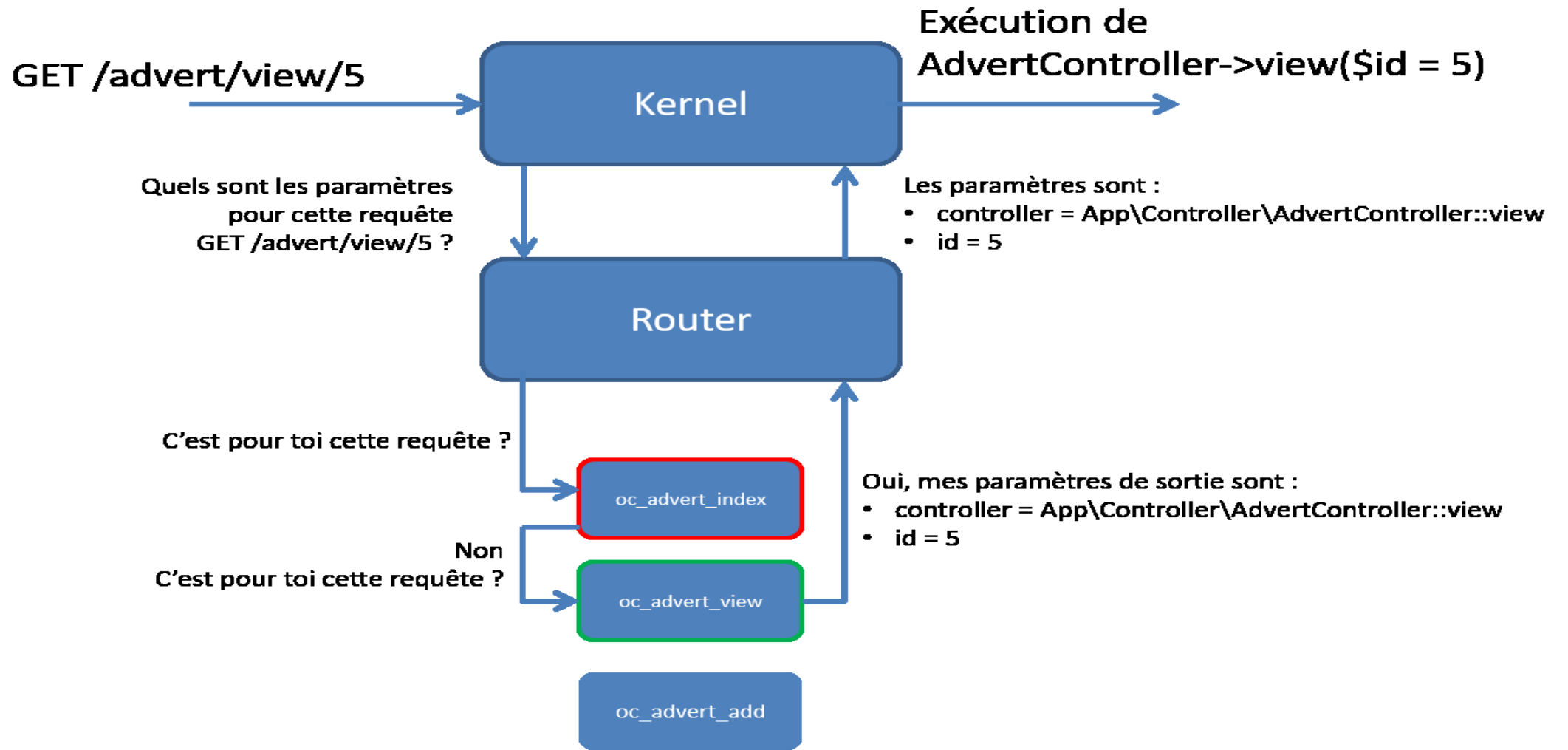
Le routage

Grâce au paramètre {id} dans le path de notre route, toutes les URL du type /advert/view/* seront gérées par cette route, par exemple :

- /advert/view/5 ,
- ou /advert/view/654,
- ou même /advert/view/azerty : on n'a pas encore dit que {id} devait être un nombre!

Les bases de symfony

Le routage



Les bases de symfony

Le routage

1. On appelle l'URL /advert/view/5.
2. Le routeur essaie de faire correspondre cette URL avec le path de la première route. Ici, /advert/view/5 ne correspond pas du tout à /advert (ligne path de la première route).
3. Le routeur passe donc à la route suivante. Il essaie de faire correspondre /advert/view/5 avec /advert/view/{id}. Nous le verrons plus loin, mais {id} est un paramètre, une sorte de joker « je prends tout ». Cette route correspond, car nous avons bien :
4. /advert/view (URL) = /advert/view (route) ;
5. 5 (URL) = {id} (route).
6. Le routeur s'arrête donc, il a trouvé sa route.
7. Il demande à la route : « Quels sont tes paramètres de sortie ? », la route répond : « Mes paramètres sont 1/ le contrôleur App\Controller\AdvertController::view, et 2/ la valeur \$id = 5. »
8. Le routeur renvoie donc ces informations au Kernel (le noyau de Symfony).
9. Le noyau exécute le bon contrôleur avec les bons paramètres !

Les bases de symfony

Exemple: Construction des routes

On souhaite avoir une URL très simple pour la page d'accueil de la partie annonces : **/advert**. Comme **/advert** est défini comme préfixe lors du chargement des routes de notre contrôleur, le path de notre route est «/». Cette page va lister les dernières annonces. Mais on veut aussi pouvoir parcourir les annonces plus anciennes, donc il nous faut une notion de page. En ajoutant le paramètre facultatif {page}, nous aurons :

/advert	page = 1
/advert/1	page = 1
/advert/2	page = 2

Pour construire cette route il nous faut utiliser les sections :

- requirements : pour ne prendre que des chiffres comme numéro de page ;
- defaults : pour rendre facultatif le numéro de page.

Les bases de symfony

Exemple: Construction des routes

```
<?php
/**
 * @Route("/{page}", name="oc_advert_index", requirements={"page" = "\d+"},
 defaults={"page" = 1})
 */
public function index()
```

Pour la page de visualisation d'une annonce, il faut bien penser au paramètre {id} qui nous servira à récupérer la bonne annonce côté contrôleur. Voici la route :

```
<?php
/**
 * @Route("/view/{id}", name="oc_advert_view", requirements={"id" = "\d+"})
 */
public function view($id)
```

Les bases de symfony

Exemple: Construction des routes

```
<?php
/**
 * @Route("/add", name="oc_advert_add")
 */
public function add()
{
}

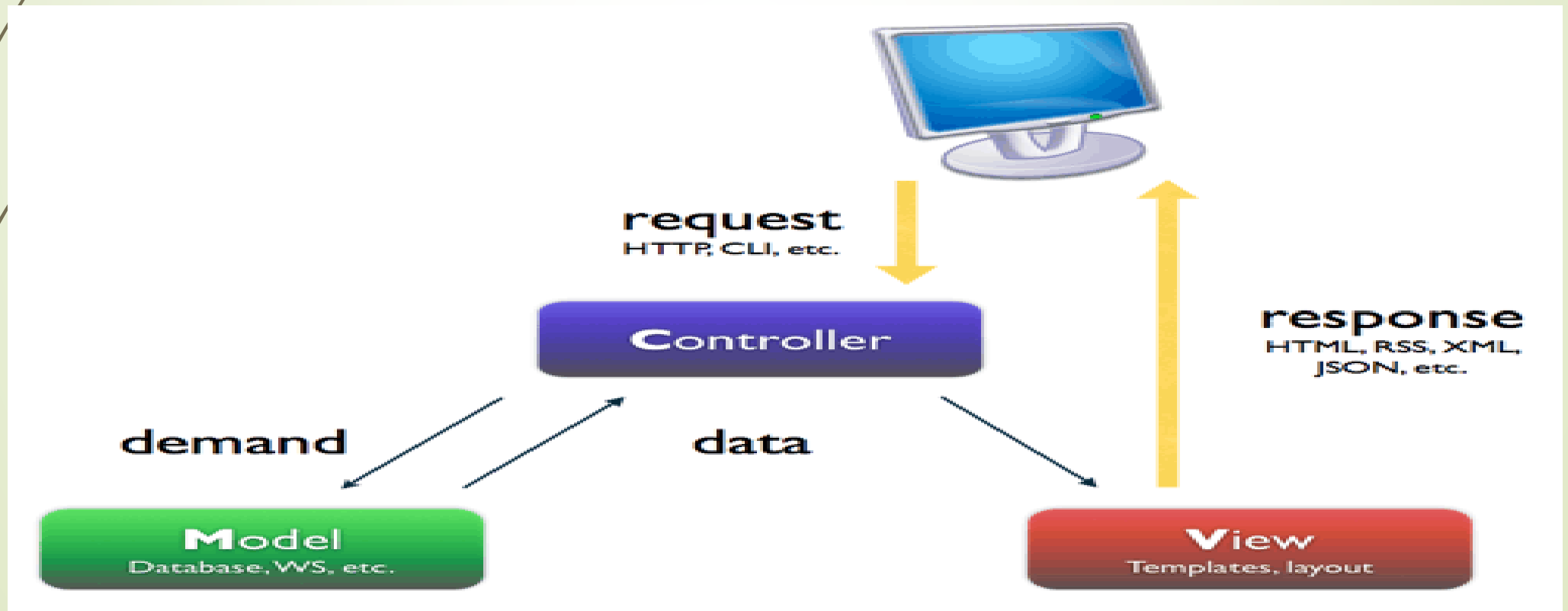
/**
 * @Route("/edit/{id}", name="oc_advert_edit", requirements={"id" = "\d+"})
 */
public function edit($id)
{
}

/**
 * @Route("/delete/{id}", name="oc_advert_delete", requirements={"id" = "\d+"})
 */
public function delete($id)
{
}
```

Les bases de symfony

L'architecture Modèle/Vue/Contrôleur

L'architecture MVC définit un cadre d'organisation de votre code en accord avec sa nature. Ce modèle permet une séparation de votre code en trois couches : Modèle, Vue et Contrôleur.



Les bases de symfony

L'architecture Modèle/Vue/Contrôleur

- ❑ La couche **Modèle** contenant le traitement logique de vos données (les accès à la base de données se trouvent dans cette couche). Vous savez déjà que symfony stocke toutes les classes et tous les fichiers relatifs au Modèle dans le répertoire **src/Entity**.
- ❑ **La Vue** est la couche où interagit l'utilisateur (un moteur de template fait parti de cette couche). Dans symfony, la couche vue est principalement faite de Templates PHP. Ces fichiers sont stockés dans les différents dossiers **templates/** comme nous le verrons plus loin.
- ❑ **Le Contrôleur** est un morceau de code qui appelle le modèle pour obtenir certaines données qu'il passe à la Vue pour le rendu au client. Quand nous avons installé symfony le premier jour, nous avons vu que toutes les requêtes étaient gérées par le contrôleur frontal **index.php**. Ce contrôleur frontal délègue le réel travail à des actions.

Les bases de symfony

Le contrôleur

- Il contient toute la logique de notre site Internet. Cependant, cela ne veut pas dire qu'il contient beaucoup de code. En fait, il ne fait qu'utiliser des services, des modèles et appeler des templates. Finalement, c'est un chef d'orchestre qui se contente de faire la liaison entre tout le monde.
- Le rôle principal du contrôleur est de retourner une réponse.
- Symfony s'est inspiré des concepts du protocole HTTP. Il existe dans Symfony les classes **Request** et **Response**. Retourner une réponse signifie donc tout simplement : instancier un objet Response, disons **\$response**, et faire un return **\$response**.
- Les objets Request et Response permettent de construire une réponse en fonction de la requête ;

Les bases de symfony

Le contrôleur

Dans une application Symfony, les contrôleurs se trouvent dans le répertoire `/src/Controller`.

Rappelez-vous : dans la route, on a dit qu'il fallait faire appel au contrôleur nommé « `App\Controller\AdvertController` ».

```
1. <?php
2. // src/Controller/AdvertController.php
3. namespace App\Controller;
4. use Symfony\Component\HttpFoundation\Response;
5. class AdvertController
6. { public function index()
7. {
8.     $content = "Notre propre Hello World !"
9.     return new Response($content);
10. }
11. }
```

Les bases de symfony

Le contrôleur

Ligne 3 : on se place dans le namespace des contrôleurs de notre application. Suivez la structure des répertoires dans lequel se trouve le contrôleur.

Ligne 4 : notre contrôleur va utiliser l'objet Response, il faut donc le définir grâce au use.

Ligne 5 : le nom de notre contrôleur respecte le nom du fichier pour que l'autoload fonctionne.

Ligne 6 : on définit la méthode index(), ce que mentionne la route.

Ligne 8 : on définit un contenu pour notre page, ici un texte simple, sans HTML.

Ligne 9 : on crée une réponse. L'argument de l'objet Response est le contenu de la page que vous envoyez au visiteur, ici « Notre propre Hello World ! ». Puis on retourne cet objet.

Les bases de symfony

Le contrôleur

➡ L'objet Request

- Pour l'exemple d'affichage d'une annonce à partir de son ID. Une requête contient des paramètres : l'id d'une annonce à afficher, le nom d'un membre à chercher dans la base de données, etc. Les paramètres sont la base de toute requête : la construction de la page à afficher dépend de chacun des paramètres en entrée.

```
<?php
// src/Controller/AdvertController.php

/**
 * @Route("/view/{id}", name="oc_advert_view", requirements={"id" = "\d+"})
 */
public function view($id)
{
    return new Response("Affichage de l'annonce d'id : ".$id);
}
```

Les bases de symfony

Le contrôleur

➡ L'objet Request

- le paramètre **{id}** de la requête est récupéré par la route, qui le transforme en argument **\$id** pour le contrôleur. Vous connaissez donc la première manière de récupérer des arguments : ceux contenus dans la route.
- En plus des paramètres de routes que nous venons de voir, vous pouvez récupérer les autres paramètres de l'UR. Prenons par exemple l'URL **/advert/view/5?tag=developer**, il nous faut bien un moyen pour récupérer ce paramètre **tag** : c'est ici qu'intervient l'objet **Request**.

Les bases de symfony

Le contrôleur

➡ L'objet Request

Pour récupérer la requête depuis un contrôleur, il faut l'injecter comme nous avons injecté des objets. Pour cela, ajoutez un argument à votre méthode avec le type hint Request comme ceci :

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
class AdvertController extends Controller
{
    /**
     * @Route("/view/{id}", name="oc_advert_view", requirements={"id" = "\d+"})
     */
    public function view($id, Request $request)
    {
        // Vous avez accès à la requête HTTP via $request
    }
}
```


Les bases de symfony

Le contrôleur

► L'objet Request

Après avoir demandé au routeur quel contrôleur exécuter, et avant de l'exécuter effectivement, il regarde si l'un des arguments de la méthode est typé avec **Request**. Si c'est le cas, il ajoute la requête aux arguments avant d'exécuter le contrôleur.

```
<?php
<?php
public function view($id, Request $request)
{
    // On récupère notre paramètre tag
    $tag = $request->query->get('tag');

    return new Response(
        "Affichage de l'annonce d'id : ".$id.", avec le tag : ".$tag
    );
} }
```

Les bases de symfony

Type de paramètres	Méthode Symfony	Méthode traditionnelle	Exemple
Variables d'URL	<code>\$request->query</code>	<code>\$_GET</code>	<code>\$request->query->get('tag')</code>
Variables de formulaire	<code>\$request->request</code>	<code>\$_POST</code>	<code>\$request->request->get('tag')</code>
Variables de cookie	<code>\$request->cookies</code>	<code>\$_COOKIE</code>	<code>\$request->cookies->get('tag')</code>
Variables de serveur	<code>\$request->server</code>	<code>\$_SERVER</code>	<code>\$request->server->get('REQUEST_URI')</code>
Variables d'entête	<code>\$request->headers</code>	<code>\$_SERVER['HTTP_*']</code>	<code>\$request->headers->get('USER_AGENT')</code>
Paramètres de route	<code>\$request->attributes</code>	n/a	On utilise <code>\$id</code> dans les arguments de la méthode, mais vous pourriez également écrire <code>\$request->attributes->get('id')</code>

Les bases de symfony

Le contrôleur

➡ L'objet Request

Heureusement, l'objet Request ne se limite pas à la récupération de paramètres. Il permet de savoir plusieurs choses intéressantes à propos de la requête en cours, voyons ses possibilités.

Récupérer la méthode de la requête HTTP

Pour savoir si la page a été récupérée via GET (clic sur un lien) ou via POST (envoi d'un formulaire), il existe la méthode **`$request->isMethod()`** :

```
<?php
if ($request->isMethod('POST'))
{
    // Un formulaire a été envoyé, on peut le traiter ici
}
>
```

Les bases de symfony

Le contrôleur

➡ L'objet Request

Savoir si la requête est une requête AJAX (Asynchronous JavaScript and XML)

Lorsque vous utiliserez AJAX dans votre site, vous aurez sans doute besoin de savoir, depuis le contrôleur, si la requête en cours est une requête AJAX ou non. Par exemple, pour renvoyer du XML ou du JSON à la place du HTML.

```
<?php
if ($request->isXmlHttpRequest())
{
    // C'est une requête AJAX, retournons du JSON, par exemple
}
>
```

Les bases de symfony

Le contrôleur

➡ L'objet Request

C'est quoi AJAX (Asynchronous JavaScript and XML) ?

- ➡ Le terme **AJAX** désigne une technologie qui s'est popularisée dans le domaine de la création de sites internet. Elle est principalement utilisée pour apporter de l'interactivité au sein des pages d'un site web tout en économisant les ressources serveur.
- ➡ En effet, AJAX permet de communiquer avec le serveur à l'aide de code Javascript en arrière-plan pendant que la page est affichée à l'écran. Ainsi le contenu de la page peut être modifié sans qu'il soit nécessaire de faire transiter et afficher la page en entier. Elle est particulièrement utilisée pour la mise à jour des formulaires et des paniers sur la plupart des sites web. C'est une technologie asynchrone : le code de la page continue de s'exécuter pendant que l'appel vers le serveur est effectué. Il faut garder à l'esprit cette information quand on utilise **AJAX**.

Les bases de symfony

Le contrôleur

➡ L'objet Request

C'est quoi JSON (*JavaScript Object Notation*) ?

- ➡ (JSON) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.
- ➡ Un document JSON comprend deux types d'éléments structurels :
 - des ensembles de paires « nom » (alias « clé ») / « valeur »
 - des listes ordonnées de valeurs
- ➡ Ces mêmes éléments représentent trois types de données :
 - des objets
 - des tableaux
 - des valeurs génériques de type tableau, objet, booléen, nombre, chaîne de caractères ou null.

Les bases de symfony

Le contrôleur

➡ L'objet Request

C'est quoi JSON (*JavaScript Object Notation*) ? : Exemple

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```

Format JSON

```
<popup>
  <menuitem value="New"
onclick="CreateNewDoc()"/>
  <menuitem value="Open"
onclick="OpenDoc()"/>
  <menuitem value="Close"
onclick="CloseDoc()"/>
</popup>
</menu>
```

Format XML

```
menu:
  id: file
  value: File
  popup:
    menuitem:
      - value: New
        onclick: CreateNewDoc()
      - value: Open
        onclick: OpenDoc()
      - value: Close
        onclick: CloseDoc()
```

Formal YAML

Les bases de symfony

Le contrôleur

➡ L'objet Response

Commençons par le dernier exemple:

Si on voudrait exécuter cette route : /genielogiciel/Foulen?classe=FlA1 &groupe=1

```
<?php
/**
 * @Route("/genielogiciel/{x}", name="oc_advert_genie")
 */
public function groupe1 (Request $req)
{
    $res=$req->query->get('classe');
    $res2=$req->query->get('groupe');
    return new Response
("salut ".$req->attributes->get('x')." vous êtes dans la classe ". $res. " et dans le groupe
".$res2);
}
```

Les bases de symfony

Le contrôleur

➡ L'objet Response

```
<?php
/**
 * @Route("/genielogiciel/{id}", name="oc_advert_genie")
 */
public function groupe1 (Environment $a, Request $req)
{
    $res=$req->query->get('classe');
    $res2=$req->query->get('groupe');
    return new Response ($a->render('groupe.html.twig',
['id' => $req->attributes->get('id'), 'classe' => $res, 'groupe'=>$res2]
));
}
```

Les bases de symfony

Le contrôleur

➡ L'objet Response

```
{# templates/groupe.html.twig #}  
  
<!DOCTYPE html>  
<html>  
<head>  
    <title>Affichage des informations de l'étudiant {{ id }}</title>  
</head>  
<body>  
    <h1>Affichage des informations de l'étudiant {{ id }}</h1>  
    <h2>Classe n°{{ classe }} !</h2>  
    <p>Numéro de groupe : {{ groupe }}</p>  
</body>  
</html>
```

Les bases de symfony

Le contrôleur

➡ L'objet Response et RedirectResponse

Pour simplifier la construction d'une réponse faisant une redirection, il existe l'objet **RedirectResponse** qui étend l'objet **Response** que nous connaissons bien, en lui ajoutant l'entête HTTP *Location* qu'il faut pour que notre navigateur comprenne qu'il s'agit d'une redirection. Cet objet prend en argument de son constructeur l'URL vers laquelle rediriger, URL que vous générez grâce au routeur.

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\RedirectResponse; // Nouveau use

class AdvertController extends Controller
{
    public function view()
    {
        $url = $this->generateUrl('oc_advert_index');
        return new RedirectResponse($url);
    }
}
```

Les bases de symfony

Le contrôleur

➡ L'objet Response et RedirectResponse

À l'image de la méthode **render**, il existe également une méthode raccourcie pour faire une redirection depuis un contrôleur, il s'agit de la méthode **redirect** qui prend en argument l'URL. L'avantage est que vous n'avez pas à rajouter le use **RedirectResponse** en début de fichier, la méthode crée la réponse elle-même :

```
<?php

public function view($id)
{
    $url = $this->generateUrl('oc_advert_index');

    return $this->redirect($url);
}
```

```
<?php

public function viewAction($id)
{
    return $this->redirectToRoute('oc_advert_index');
}
```

Les bases de symfony

Le contrôleur

➡ L'objet Response

Lorsque vous retournez autre chose que du HTML, il faut que vous changiez l'en-tête **Content-Type** de la réponse. Prenons l'exemple suivant : vous recevez une requête AJAX et souhaitez retourner un tableau en JSON :

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{ public function view($id)
{ // Créons nous-mêmes la réponse en JSON, grâce à la fonction json_encode()
$response = new Response(json_encode(['id' => $id]));
// Ici, nous définissons le Content-Type pour dire au navigateur que l'on renvoie du JSON et non du HTML
$response->headers->set('Content-Type', 'application/json');
return $response; }
}
```


Les bases de symfony

Le contrôleur

➔ L'objet JsonResponse

Il existe l'objet **JsonResponse** qui ne fait rien d'autre que ce qu'on vient de faire : encoder son argument grâce à la méthode **json_encode** , puis définir le bon Content-Type . Le code suivant montre très concrètement comment on aurait pu écrire l'exemple précédent.

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{ public function view($id)
  {return new JsonResponse(['id' => $id]);}
}
```

Les bases de symfony

Le contrôleur

► Manipuler la session

Une des actions classiques d'un contrôleur, c'est de manipuler la session. Dans Symfony, il existe un objet Session qui permet de gérer la session, il se récupère comme l'objet Twig, en argument de la méthode.

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Session\SessionInterface; // Nouveau use
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{
    // On injecte la session avec SessionInterface
    public function viewAction($id, SessionInterface $session)
    {
        // On récupère le contenu de la variable userId
        $userId = $session->get('userId');
        // On définit une nouvelle valeur pour cette variable userId
        $session->set('userId', 91);
        // On n'oublie pas de renvoyer une réponse
        return new Response("<body>Je suis une page de test, je n'ai rien à dire</body>");
    }
}
```

Les bases de symfony

Le contrôleur

► Manipuler la session

La session se lance automatiquement dès que vous vous en servez. Voyez par exemple à la figure suivante ce que le Profiler me dit sur une page où je n'utilise pas la session.

Le Profiler nous donne même les informations sur la date de création de la session, etc.

The screenshot shows the Symfony Profiler interface in a web browser. The address bar displays the URL `127.0.0.1:8000/_profiler/316f9a`. The page title is "Symfony Profiler". The main content area shows the URL `http://127.0.0.1:8000/sess` with the following details: Method: GET, HTTP Status: 200, IP: 127.0.0.1, Profiled on: Fri, 28 Feb 2020 23:27:21 +0000, and Token: 316f9a.

The left sidebar contains a menu with the following items: Last 10, Latest, Search, Request / Response, Performance, Validator, Forms, Exception, Logs, Events, Routing, Cache, Translation, and Security.

The main content area displays the "AdvertController :: viewAction" page. The "Session" tab is selected, showing the following session metadata:

Key	Value
Created	"Fri, 28 Feb 20 23:07:28 +0000"
Last used	"Fri, 28 Feb 20 23:27:21 +0000"
Lifetime	"0"

Below the session metadata, the "Session Attributes" section shows the following attribute:

Attribute	Value
userId	91

Les bases de symfony

Le contrôleur

► Manipuler la session

Un autre outil très pratique offert par cet objet Session est ce que l'on appelle les « messages flash ». Un terme précis pour désigner en réalité une variable de session qui ne dure que le temps d'une seule page.

C'est une astuce utilisée pour les formulaires par exemple :

- la page qui traite le formulaire définit un message flash (« Annonce bien enregistrée » par exemple) puis redirige vers la page de visualisation de l'annonce nouvellement créée.
- Sur cette page, le message flash s'affiche, et est détruit de la session. Alors si l'on change de page ou qu'on l'actualise, le message flash ne sera plus présent.

Les bases de symfony

Le contrôleur

➡ Manipuler la session

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Session\SessionInterface; // Nouveau use
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{
    public function view(Request $request)
    {
        return $this->render('Advert/view.html.twig', ['id' => $request->query->get('id'), ]);
    }
    public function add()
    {
        // Bien sûr, cette méthode devra réellement ajouter l'annonce Mais faisons comme si c'était le cas
        $this->addFlash('info', 'Annonce bien enregistrée');
        // Le « flashBag » est ce qui contient les messages flash dans la session Il peut bien sûr contenir plusieurs messages :
        $this->addFlash('info', 'Oui oui, elle est bien enregistrée !');
        // Puis on redirige vers la page de visualisation de cette annonce
        return $this->redirectToRoute('oc_advert_view', ['id' => 5]);
    }
}
```

Les bases de symfony

Le contrôleur

➡ Manipuler la session

```
{# templates/Advert/view.html.twig #}
<!DOCTYPE html>
<html>
<head>
  <title>Affichage de l'annonce {{ id }}</title>
</head>
<body>
<h1>Affichage de l'annonce n°{{ id }} !</h1>
<div>
  {# On affiche tous les messages flash dont le nom est « info » #}
  {% for message in app.flashes('info') %}
    <p>Message flash : {{ message }}</p>
  {% endfor %}
</div> <p>
  Ici nous pourrions lire l'annonce ayant comme id : {{ id }}<br />
  Mais pour l'instant, nous ne savons pas encore le faire, cela viendra !
</p> </body> </html>
```


Les bases de symfony

Le contrôleur

► Manipuler la session

- La variable Twig `{{ app }}` est une variable globale, disponible partout dans vos vues. Elle contient quelques variables et méthodes utiles, nous le verrons, dont la méthode « flashes » que nous venons d'utiliser via `{{ app.flashes('xxx') }}`.
- Pour tester l'interprétation des messages
 - Demander la route appelant la fonction **add**
 - Actualiser la page affichée!!!

Les bases de symfony

Le contrôleur

➤ Gestion des exceptions

- Une exception est une alerte lancée lors de l'exécution du code, pour indiquer que quelque chose ne s'est pas passé comme prévu.
- Une fois que cette alerte est levée, il faut bien en faire quelque chose. Soit je la laisse sous le tapis et je la passe sous silence, soit je la gère correctement pour que mon application continue de fonctionner normalement même après cette erreur.
- PHP 7 apporte un changement à la façon dont les erreurs sont rapportées. Désormais la plupart des erreurs sont rapportées en lançant des exceptions.
- Le composant **ErrorHandler** fournit des outils pour gérer les erreurs et faciliter le débogage du code PHP.

Les bases de symfony

Le contrôleur

➡ Gestion des exceptions

- Chaque exception doit être jetée dans la pile d'exécution jusqu'à rencontrer un des cas suivants :
 - Si elle rencontre un bloc **catch** qui supporte ce type d'erreur ;
 - Si un gestionnaire d'exception est configuré via **set_exception_handler()** ;
 - Sinon l'exception sera convertie en erreur **FATAL** et sera traitée par le système traditionnel.
- **Throwable** est une interface PHP 7 qui représente une erreur dans le script.
- **Errors** et **Exceptions** sont les deux types de bases qui l'implémentent.
 - ❖ **Exception** est la classe de base de toutes les exceptions utilisateurs.
 - ❖ **Error** est la classe de base de toutes les erreurs internes de PHP.

Les bases de symfony

Le contrôleur

➤ Gestion des exceptions

- La classe **ErrorHandler** capture les erreurs PHP et les exceptions PHP non capturées et les transforme en objets **ErrorException** de PHP, à l'exception des erreurs PHP fatales, qui sont transformées en objets **FatalError** de Symfony.
- Il existe quelques bonnes pratiques sur la création des exceptions. Je vais m'intéresser à deux en particulier : le nommage et le contexte.
 - Pour lancer une exception, il suffit d'utiliser le mot clé **throw**.
 - Pour attraper et gérer l'exception, il faut utiliser la structure **try catch**.

Les bases de symfony

Le contrôleur

► Lancer une exception

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{
    /**
     * @Route("/{page}", name="oc_advert_index", requirements={"page" = "\d+"}, defaults={"page" = 1})
     */
    public function index($page)
    {
        // On ne sait pas combien de pages il y a Mais on sait qu'une page doit être supérieure ou égale à 1
        if ($page < 1) {
            // On déclenche une exception NotFoundHttpException, cela va afficher une page d'erreur 404
            throw $this->createNotFoundException('Page "'.$page.'" inexistante.');
        }
        // Ici, on récupérera la liste des annonces, puis on la passera au template
        return $this->render('index.html.twig'); }
}
```

Les bases de symfony

Le contrôleur

► Attraper une exception

```
<?php
// src/Controller/AdvertController.php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
class AdvertController extends AbstractController
{
    /**
     * @Route("/{page}", name="oc_advert_index", requirements={"page" = "\d+"}, defaults={"page" = 1})
     */
    public function index($page){
        Try{
            if ($page < 1) {
                // On attrape une exception NotFoundHttpException, cela va afficher une page d'erreur 404
            }
            Catch (NotFoundHttpException $e){
                if (404 === $e->getResponse()->getStatusCode()){.....}
                // Ici, on récupérera la liste des annonces, puis on la passera au template
                return $this->render('index.html.twig'); }
        }
    }
}
```


Les bases de symfony

Les vues

- Constituent la partie présentation de l'application
- Utilisent le moteur de templates (modèles) Twig
- Reçoivent des données du contrôleur
- Peuvent utiliser des structures algorithmiques
- Ne contiennent aucun code PHP
- Utilisent le pseudo-langage Twig à base de { }
- Les templates Twig peuvent être hérités ou inclus pour un meilleur découpage
- Classiquement, 3 niveaux d'héritage :
application → section → page

Les bases de symfony

Les vues

TWIG

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue dans Symfony!</title>
  </head>
  <body>
    <h1>{{ titre_page }}</h1>

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

PHP

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue dans Symfony !</title>
  </head>
  <body>
    <h1><?php echo $titre_page; ?></h1>

    <ul id="navigation">
      <?php foreach ($navigation as $item) { ?>
        <li>
          <a href="<?php echo $item->getHref();
?>"><?php echo $item->getTitre(); ?></a>
        </li>
      <?php } ?>
    </ul>
  </body>
</html>
```

Les bases de symfony

Description	Exemple Twig	Équivalent PHP
Afficher une variable	Pseudo : {{ pseudo }}	Pseudo : <?php echo \$pseudo; ?>
Afficher l'index d'un tableau	Identifiant : {{ user['id'] }}	Identifiant : <?php echo \$user['id']; ?>
Afficher l'attribut d'un objet, dont le getter respecte la convention \$objet->getAttribut()	Identifiant : {{ user.id }}	Identifiant : <?php echo \$user->getId(); ?>
Afficher une variable en lui appliquant un filtre. Ici, « upper » met tout en majuscules :	Pseudo en majuscules : {{ pseudo upper }}	Pseudo en lettre majuscules : <?php echo strtoupper(\$pseudo); ?>
Utiliser un filtre avec des arguments. Attention, il faut que <code>date</code> soit un objet de type <code>Datetime</code> ici.	Date : {{ date date('d/m/Y') }}	Date : <?php echo \$date->format('d/m/Y'); ?>
Concaténer	Identité : {{ nom ~ " " ~ prenom }}	Identité : <?php echo \$nom.' '.\$prenom; ?>

Les bases de symfony

Le fonctionnement de la syntaxe **{{ objet.attribut }}** est un peu plus complexe qu'il n'en a l'air. Elle ne fait pas seulement **\$objet->getAttribut()**. En réalité, voici ce qu'elle fait exactement :

- i. Elle vérifie si objet est un **tableau**, et si attribut en est un index valide. Si c'est le cas, elle affiche **objet['attribut']**.
- ii. Sinon, et si objet est un **objet**, elle vérifie si attribut en est un **attribut** valide (public donc). Si c'est le cas, elle affiche **objet->attribut**.
- iii. Sinon, et si objet est un **objet**, elle vérifie si attribut() en est une **méthode** valide (publique donc). Si c'est le cas, elle affiche **objet->attribut()**.
- iv. Sinon, et si objet est un objet, elle vérifie si **getAttribut()** en est une méthode valide. Si c'est le cas, elle affiche **objet->getAttribut()**.
- v. Sinon, et si objet est un objet, elle vérifie si **isAttribut()** en est une méthode valide. Si c'est le cas, elle affiche **objet->isAttribut()**.
- vi. Sinon, elle n'affiche rien et retourne **null**.

Les bases de symfony

Filtre	Description	Exemple Twig
<u>upper</u>	Met toutes les lettres en majuscules.	<code>{{ var upper }}</code>
<u>striptags</u>	Supprime toutes les balises XML.	<code>{{ var striptags }}</code>
<u>date</u>	Formate la date selon le format donné en argument. La variable en entrée doit être une instance de Datetime.	<code>{{ date date('d/m/Y') }}</code> Date d'aujourd'hui : <code>{{ "now" date('d/m/Y') }}</code>
<u>format</u>	Insère des variables dans un texte, équivalent à <u>printf</u> .	<code>{{ "Il y a %s pommes et %s poires" format(153, nb_poires) }}</code>
<u>length</u>	Retourne le nombre d'éléments du tableau, ou le nombre de caractères d'une chaîne.	Longueur de la variable : <code>{{ texte length }}</code> Nombre d'éléments du tableau : <code>{{ tableau length }}</code>

Les bases de symfony

Symfony enregistre par défaut une variable globale **{{ app }}** dans Twig pour nous faciliter la vie. Voici la liste de ses attributs, qui sont donc disponibles dans tous vos templates :

Variable	Description
{{ app.request }}	La requête « request » qu'on a vue au chapitre précédent sur les contrôleurs.
{{ app.session }}	Le service « session » qu'on a vu également au chapitre précédent.
{{ app.environment }}	L'environnement courant : « dev », « prod », et ceux que vous avez définis.
{{ app.debug }}	True si le mode debug est activé, False sinon.
{{ app.user }}	L'utilisateur courant, que nous verrons également plus loin dans ce cours.
{{ app.flashes('xxx') }}	Récupère les messages flashes stockés dans la session.

Les bases de symfony

On peut enregistrer nos propres variables globales, pour qu'elles soient accessibles depuis tous nos templates, au lieu de les injecter à chaque fois depuis le contrôleur. Pour cela, il faut éditer le fichier de configuration de l'application, comme suit

```
twig:  
  default_path: '%kernel.project_dir%/templates'  
  debug: '%kernel.debug%'  
  strict_variables: '%kernel.debug%'  
  exception_controller: null  
  globals:  
    webmanager: 'Zarrouk Elyes'
```

Ainsi, la variable {{ webmanager }} sera injectée dans tous les templates, et donc utilisable comme ceci :

```
<footer>Responsable du site : {{ webmanager }}.</footer>
```

Les bases de symfony

Pour ce genre de valeurs paramétrables, la bonne pratique est de les définir non pas directement dans les fichiers de configuration situés dans le répertoire **config** , mais dans le fichier des paramètres d'environnement, à savoir le fichier **.env** à la racine du projet.

```
# .env  
  
# Rajoutez ceci à la fin du fichier  
WEBMASTER=moi-même
```

Ainsi, il faut modifier le fichier de configuration **Twig.yaml** en utilisant les variables d'environnement

```
# config/packages/twig.yaml  
  
twig:  
  globals:  
    webmaster: "%env(WEBMASTER)%"
```

Les bases de symfony

Structures de contrôle

► Condition : {% if %}

TWIG

```
{% if membre.age < 12 %}  
    Il faut avoir au moins 12 ans pour ce film.  
{% elseif membre.age < 18 %}  
    OK bon film.  
{% else %}  
    Un peu vieux pour voir ce film non ?  
{% endif %}
```

► Structure {% set %}

TWIG

```
{% set foo = 'bar' %}
```

PHP

```
<?php  
if($membre->getAge() < 12) { ?>  
    Il faut avoir au moins 12 ans pour ce film.  
<?php } elseif($membre->getAge() < 18) { ?>  
    OK bon film.  
<?php } else { ?>  
    Un peu vieux pour voir ce film non ?  
<?php } ?>
```

PHP

```
<?php $foo = 'bar'; ?>
```

Les bases de symfony

Structures de contrôle

► Condition : {% for %}

```
<ul>
{% for membre in liste_membres %}
  <li>{{ membre.pseudo }}</li>
{% else %}
  <li>Pas d'utilisateur trouvé.</li>
{% endfor %}
</ul>
```

TWIG

```
<select>
{% for valeur, option in liste_options %}
  <option value="{{ valeur }}">{{ option }}
</option>
{% endfor %}
</select>
```

TWIG

```
PHP

<ul>
<?php if(count($liste_membres) > 0) {
  foreach($liste_membres as $membre) {
    echo '<li>'.$membre->getPseudo().'</li>';
  }
} else { ?>
  <li>Pas d'utilisateur trouvé.</li>
<?php } ?>
</ul>
```

```
PHP

<?php
foreach($liste_options as $valeur => $option) {
  // ...
}
```

Les bases de symfony

Structures de contrôle

► Condition : {% for %}

Variable	Description
{{ loop.index }}	Le numéro de l'itération courante (en commençant par 1).
{{ loop.index0 }}	Le numéro de l'itération courante (en commençant par 0).
{{ loop.revindex }}	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 1).
{{ loop.revindex0 }}	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 0).
{{ loop.first }}	True si c'est la première itération, false sinon.
{{ loop.last }}	True si c'est la dernière itération, false sinon.
{{ loop.length }}	Le nombre total d'itérations dans la boucle.

Les bases de symfony

Structures de contrôle

► Condition : {% if %}

TWIG

```
{% if membre.age < 12 %}  
    Il faut avoir au moins 12 ans pour ce film.  
{% elseif membre.age < 18 %}  
    OK bon film.  
{% else %}  
    Un peu vieux pour voir ce film non ?  
{% endif %}
```

► Structure {% set %}

```
{% set foo = 'bar' %}
```

TWIG

► Structure defined

```
{% if var is defined %} ... {% endif %}
```

TWIG

PHP

```
<?php  
  
if($membre->getAge() < 12) { ?>  
    Il faut avoir au moins 12 ans pour ce film.  
<?php } elseif($membre->getAge() < 18) { ?>  
    OK bon film.  
<?php } else { ?>  
    Un peu vieux pour voir ce film non ?  
}
```

```
?php $foo = 'bar'; ?>
```

PHP

```
<?php if(isset($var)) { }?>
```

PHP

Les bases de symfony

Hériter et inclure des templates

- Le principe est simple : vous avez un template père qui contient le design de votre site ainsi que quelques blocs (« blocks » en anglais) et des templates fils qui vont remplir ces blocs. Les fils vont donc venir hériter du père en remplaçant certains éléments par leur propre contenu.
- L'avantage est que les templates fils peuvent modifier plusieurs blocs du template père. Avec la technique des **include()**, un template inclus ne pourra pas modifier le template père dans un autre endroit que là où il est inclus !
- Les blocs classiques sont le centre de la page et le titre. Mais en fait, c'est à vous de les définir ; vous en ajouterez donc autant que vous voudrez.

Les bases de symfony

Héritage des templates

- Voici un exemple de template père **templates/layout.html.twig** et un des templates fils **templates/Advert/index.html.twig**

```
{# templates/layout.html.twig #}
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <title>
    {% block title %}Plateforme d'annonces{% endblock %}
  </title>
</head>
<body>
  {% block body %}
  {% endblock %}
</body>
</html>
```

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

{% extends "layout.html.twig" %}

{% block title %}{{ parent() }} – Index
{% endblock %}

{% block body %}
  Notre plateforme est un peu vide pour le
  moment, mais cela viendra !
{% endblock %}
```

Les bases de symfony

Inclusion des templates

- En utilisant la fonction **include()** on peut intégrer un fichier dans un autre:

```
{# templates/Advert/add.html.twig #}
{% extends "layout.html.twig" %}
{% block body %}
    <h2>Ajouter une annonce</h2>
    {{ include("Advert/_form.html.twig") }}

    <p>
        Attention : cette annonce sera ajoutée
        directement
        sur la page d'accueil après validation du
        formulaire.
    </p>

{% endblock %}
```

```
{# templates/Advert/_form.html.twig #}
<h3>Formulaire d'annonce</h3>
<div class="well">
    <form method="POST" action="#">
        <p>Nom: <input type="text"
name="nom_annonce"/></p>
        <p>Login: <input type="text"
name="type"/></p>
        <input type="submit" name="valider"/>
    </form>
</div>
```

Les bases de symfony

Inclusion de contrôleurs

- Du côté du template qui fait l'inclusion, à la place de la fonction **{{ include() }}**, il faut utiliser la fonction **{{ render() }}**.
- **Cas d'utilisation**
 - Supposons que le modèle affiche les trois articles les plus récents dans un site des annonces. Pour ce faire, il doit effectuer une requête de base de données pour obtenir ces articles.
 - Lorsque vous utilisez la fonction **include ()**, vous devez effectuer la même requête de base de données dans chaque page qui inclut le fragment. Ce n'est pas très pratique.
 - Une meilleure alternative est d'incorporer le résultat de l'exécution d'un contrôleur avec les fonctions **Twig** de **render ()** de **controller ()**.

Les bases de symfony

Les étapes d'Inclusion de contrôleurs

- Au premier lieu, on doit créer la fonction qui renvoie les articles les plus récents

```
namespace App\Controller;
class AdvertController extends AbstractController
{
    public function recentAdverts($max = 3)
    { // renvoie les récentes annonces(ex: requête de BD)
        $listAdverts = array(
            array('id' => 2, 'title' => 'Recherche développeur Symfony'),
            array('id' => 5, 'title' => 'Mission de webmaster'),
            array('id' => 9, 'title' => 'Offre de stage webdesigner')
        );
        return $this->render('recent_adverts.html.twig', [ 'articles'=>
            $listAdverts ]); } }
```

Les bases de symfony

Les étapes d'Inclusion de contrôleurs

- Deuxièmement, on doit créer le fichier **menu.html.twig** qui permette d'afficher les 3 dernières récentes annonces

```
{% extends "layout2.html.twig" %}
{% block body %}

    {% for advert in listAdverts %}
        <li>
            <a href="{{ path('oc_view', {'id': advert.id}) }}">
                {{ advert.title }}
            </a>
        </li>
    {% endfor %}
</ul>

{% endblock %}
```


Les bases de symfony

Les étapes d'Inclusion de contrôleurs

- Finalement, on doit créer le fichier template **Layout.html.twig** qui permettra de s'orienter à chaque fois vers le contrôleur pour récupérer le résultat de la requête sans l'exécuter à chaque appel:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <title>{% block title %}Plateforme d'annonces{% endblock %}</title>
</head> <body>
{% block body %}

    <div id="menu">
        {{ render(controller("App\\Controller\\AdvertController: recentAdverts ")) }}
    </div>
{% endblock %}

</body>
</html>
```

Les bases de symfony

Les services

- Un service est simplement un objet PHP qui remplit une fonction, associé à une configuration pouvant être accessible depuis n'importe où dans votre code.
- Cette fonction peut être simple : envoyer des e-mails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.
- Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes et leur configuration).
- Un service est avant tout une simple classe.
- Quant à la configuration d'un service, c'est juste un moyen de l'enregistrer dans le conteneur de services. On lui donne un nom, on précise quelle est sa classe, et ainsi le conteneur a la carte d'identité du service.

```
>php bin/console debug:autowiring
```

Les bases de symfony

La programmation orientée services

- le concept de service est un bon moyen d'éviter d'utiliser trop souvent à mauvais escient le pattern singleton (utiliser une méthode statique pour récupérer l'objet depuis n'importe où).
- L'avantage de réfléchir sur les services est que cela force à bien séparer chaque fonctionnalité de l'application.
- Comme chaque service ne remplit qu'une seule et unique fonction, ils sont facilement réutilisables.
- Vous pouvez surtout facilement les développer, les tester et les configurer puisqu'ils sont assez indépendants.
- Cette façon de programmer est connue sous le nom d'architecture orientée services, et n'est pas spécifique à Symfony ni au PHP.

Les bases de symfony

Le conteneur de services

- L'intérêt réel des services réside dans leur association avec le conteneur de services.
- Ce conteneur de services est une sorte de super-objet qui gère tous les services. Ainsi, pour accéder à un service, il faut passer par le conteneur.
- L'objectif est de simplifier au maximum la récupération des services depuis votre code à vous (depuis le contrôleur ou autre).
- Vous demandez au conteneur un certain service en l'appelant par son nom, et le conteneur s'occupe de tout pour vous retourner le service demandé.

➤ Pour afficher la liste des conteneurs:

```
>php bin/console debug:container
```

Les bases de symfony

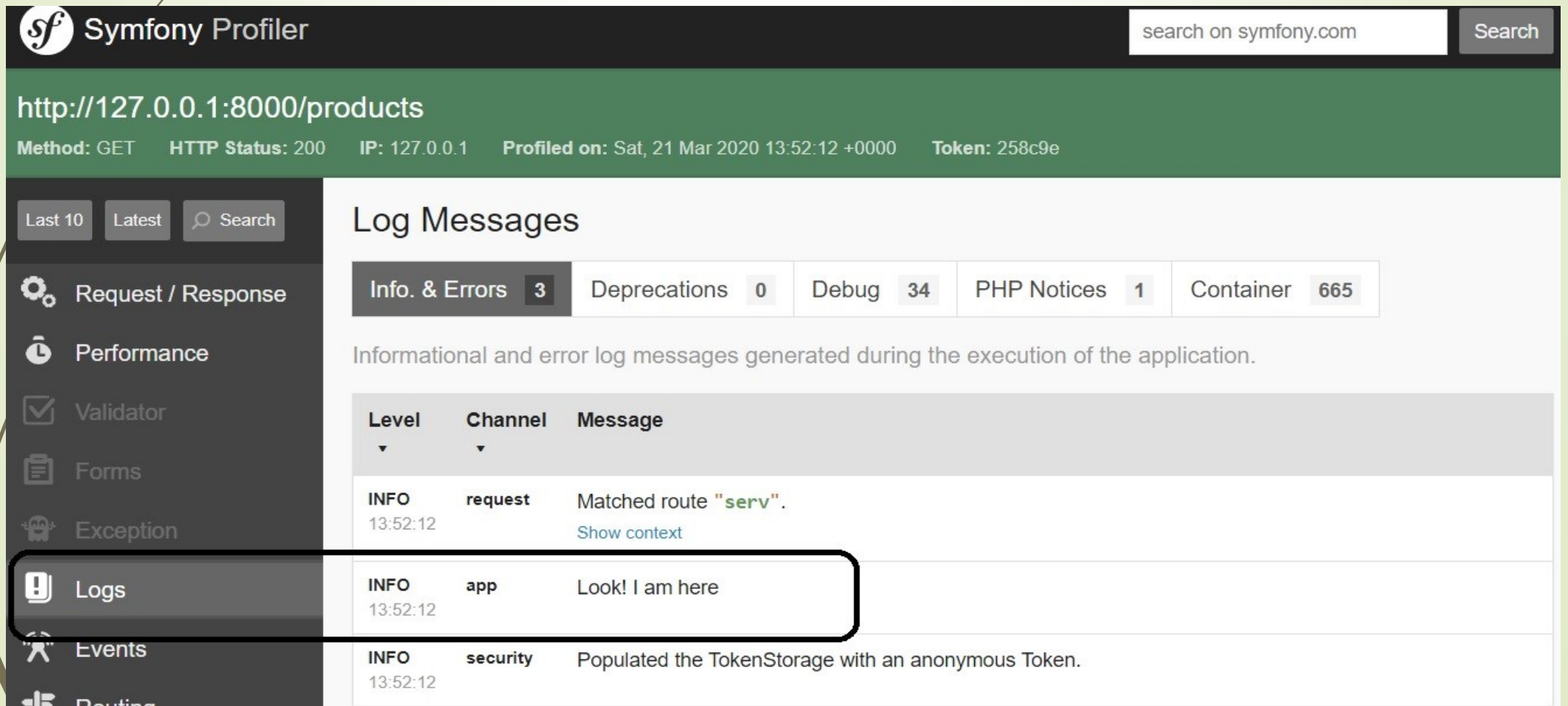
Utilisation des services:

Exemple: Service de logging

```
class ServiceController extends AbstractController
{
    /**
     *
     * @Route("/testservice", name="oc_service")
     */
    public function list(Environment $e, LoggerInterface $logger)
    {
        $logger->info('Look! I am here');
        return new Response ($e->render('serv.html.twig', ['controller_name'=>'ServiceController']));
    }
}
```

Les bases de symfony

Exemple: utilisation du service de logging



The screenshot displays the Symfony Profiler interface. At the top, the Symfony logo and 'Symfony Profiler' text are on the left, and a search bar with 'search on symfony.com' and a 'Search' button is on the right. Below this, a green bar shows the URL 'http://127.0.0.1:8000/products', the method 'GET', HTTP status '200', IP '127.0.0.1', profiled on date 'Sat, 21 Mar 2020 13:52:12 +0000', and token '258c9e'. A left sidebar contains navigation links: 'Last 10', 'Latest', 'Search', 'Request / Response', 'Performance', 'Validator', 'Forms', 'Exception', 'Logs' (highlighted with a black box), 'Events', and 'Routing'. The main area is titled 'Log Messages' and features a summary bar with 'Info. & Errors' (3), 'Deprecations' (0), 'Debug' (34), 'PHP Notices' (1), and 'Container' (665). Below this, a description states: 'Informational and error log messages generated during the execution of the application.' A table follows with columns 'Level', 'Channel', and 'Message'. The table contains three entries: 1. Level: INFO, Channel: request, Message: 'Matched route "serv".' with a 'Show context' link. 2. Level: INFO, Channel: app, Message: 'Look! I am here' (this row is highlighted with a black box). 3. Level: INFO, Channel: security, Message: 'Populated the TokenStorage with an anonymous Token.'

Symfony Profiler

search on symfony.com Search

http://127.0.0.1:8000/products

Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Sat, 21 Mar 2020 13:52:12 +0000 Token: 258c9e

Last 10 Latest Search

Request / Response

Performance

Validator

Forms

Exception

Logs

Events

Routing

Log Messages

Info. & Errors 3 Deprecations 0 Debug 34 PHP Notices 1 Container 665

Informational and error log messages generated during the execution of the application.

Level	Channel	Message
INFO 13:52:12	request	Matched route "serv". Show context
INFO 13:52:12	app	Look! I am here
INFO 13:52:12	security	Populated the TokenStorage with an anonymous Token.

Les bases de symfony

Creation d'un service de notification

1. Création d'un contrôleur

```
>php bin/console make:controller NoteController
```

2. Création d'un service

```
<?php
namespace App\Service;
class Message
{
    public function getMessage($note, $student)
    {
        $messages = 'Hi '.$student.' your note is '.$note;

        return $messages;
    }
}
```

Les bases de symfony

Creation d'un service de notification

3. Création de la fonction appelant le service

```
use App\Service\Message;

class NoteController extends AbstractController
{
    /**
     * @Route("/note/add", name="add_note")
     */
    public function add(Message $message)
    {
        $student_note=$message->getMessage('20','Elyes');
        $this->addFlash('success', $student_note);
        return $this->render('note/index.html.twig', [
            'controller_name' => 'NoteController',
        ]);
    }
}
```

Les bases de symfony

http://127.0.0.1:8000/note/add

Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Sun, 12 Apr 2020 17:47:34 +0000 Token: fe246b

Last 10

Latest

Search



Request / Response



Performance



Validator



Forms



Exception



Logs



Events

NoteController :: add

Request

Response

Cookies

Session

Flashes

Server Parameters

Flashes

Key	Value
success	["Hi Elyes your note is 20"]

Les bases de symfony

Injection d'un service dans un autre

Nous allons essayer d'injecter le service de Logging dans le service Message

```
<?php
namespace App\Service;
use Psr\Log\LoggerInterface;
class Message
{
    private $logger;
    //création d'un constructeur
    //Injection de dépendances
    public function __construct(LoggerInterface $logger)
    {
        $this->logger=$logger;
    }
    public function getMessage($note, $student)
    {
        $messages = 'Hi '.$student.' your note is '.$note;
        $this->logger->INFO($messages);
        return $messages;
    }
}
```

Les bases de symfony

Injection d'un service dans un autre

http://127.0.0.1:8000/note/add

Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Sun, 19 Apr 2020 16:01:56 +0000 Token: 4b480f

Last 10 Latest Search

Request / Response

Performance

Validator

Forms

Exception

Logs

Events

Routing

Log Messages

Info. & Errors 3

Deprecations 0

Debug 34

PHP Notices 1

Container 673

Informational and error log messages generated during the execution of the application.

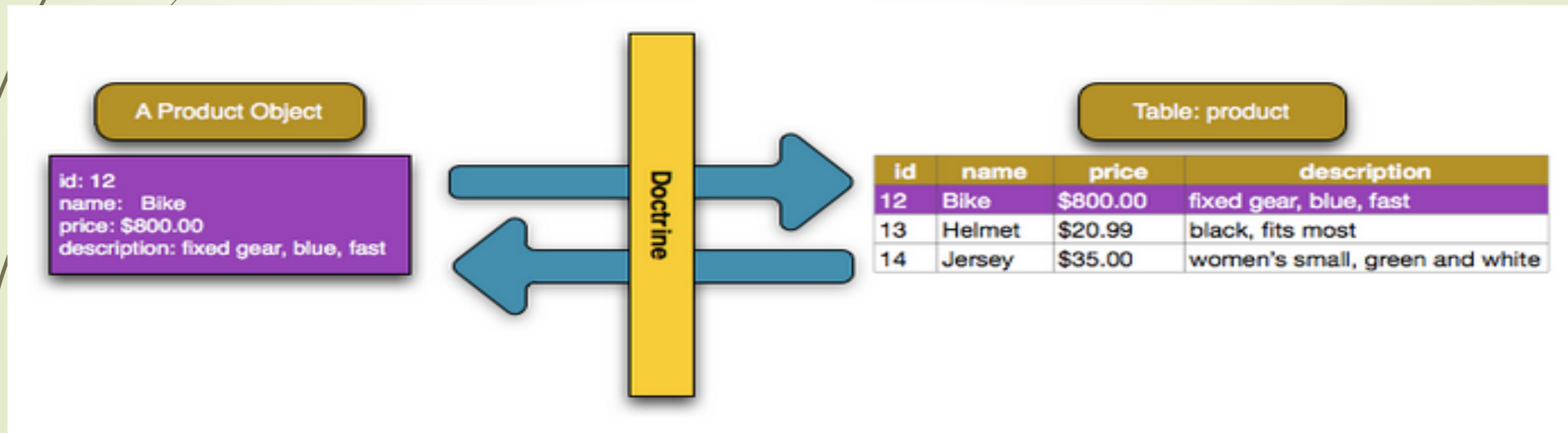
Level	Channel	Message
INFO 16:01:55	request	Matched route "add_note". Show context
INFO 16:01:55	app	Hi Elyes your note is 20
INFO 16:01:56	security	Populated the TokenStorage with an anonymous Token.



La couche métier

La couche métier: Les entités

- Une entité est, du point de vue PHP, un simple objet.
- L'entité est une classe qui possède des attributs et se situe dans /Entity
- En symfony, l'entité est une classe PHP qui peut être mappée en une table de base de donnée à travers le service **Doctrine**



La couche métier: Les entités

ORM : Object-Relational Mapping

- Couche d'abstraction d'accès aux données
- Ne plus écrire de requêtes mais des objets
- Lazy loading (chargement paresseux)
- Décrire les relations entre objets
 - One-To-One
 - Many-To-One
 - Many-To-Many
 - Many-To-Many avec attributs
- Gestion personnalisée des accès si nécessaire

Classe PHP ↔ table du SGBD

Propriétés de l'instance ↔ colonnes de la table

La couche métier: Les entités

Installation Doctrine

```
> composer require symfony/orm-pack  
> composer require --dev symfony/maker-bundle
```

Configuration de la Base de données (.env)

```
1 # .env (or override DATABASE_URL in .env.local to avoid committing your changes)  
2  
3 # customize this line!  
4 DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"  
5  
6 # to use sqlite:  
7 # DATABASE_URL="sqlite:///kernel.project_dir%/var/app.db"
```

Création de la Base de données

```
> php bin/console doctrine:database:create
```

La couche métier: Les entités

Création d'une entité

```
C:\wamp64\www\firstapp>php bin/console make:entity

Class name of the entity to create or update (e.g. FierceJellybean):
> Produit

created: src/Entity/Produit.php
created: src/Repository/ProduitRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> id

[ERROR] The "id" property already exists.

New property name (press <return> to stop adding fields):
> nom

Field type (enter ? to see all types) [string]:
>

Field length [255]:
> 120

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

updated: src/Entity/Produit.php
```

La couche métier: Les entités

Migration

- Il s'agit de tirer parti du Doctrine Migrations Bundle qui est déjà installé
- Vérification et préparation de l'environnement de gestion de Base de données selon la configuration préétablie
- La migration nécessite la création de la Base de donnée auparavant.

```
C:\wamp64\www\firstapp>php bin/console doctrine:database:create  
Created database `firstapp` for connection named default
```

```
C:\wamp64\www\firstapp>php bin/console make:migration
```

Success!

```
Next: Review the new migration "src/Migrations/Version20200519173750.php"  
Then: Run the migration with php bin/console doctrine:migrations:migrate  
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

La couche métier: Les entités

Exécuter les migrations

- Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés sur votre base de données.
- Vous devez exécuter cette commande sur la production lors de votre déploiement pour maintenir votre base de données de production à jour.

```
C:\wamp64\www\firstapp>php bin/console doctrine:migrations:migrate
```

Application Migrations

```
WARNING! You are about to execute a database migration that could result in schema changes and data loss. Are you sure you wish to continue? (y/n)y
```

```
Migrating up to 20200519173750 from 0
```

```
++ migrating 20200519173750
```

```
-> CREATE TABLE produit (id INT AUTO_INCREMENT NOT NULL, nom VARCHAR(120) DEFAULT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB
```

```
++ migrated (took 1302.1ms, used 18M memory)
```

```
-----
```

```
++ finished in 1390.6ms
```

```
++ used 18M memory
```

```
++ 1 migrations executed
```

```
++ 1 sql queries
```


La couche métier: Les entités

Mise à jour des entités

- En cas de mise des entités (attributs et méthodes) il devient indispensable de migrer cette modification vers la Base de données.

```
C:\wamp64\www\firstapp>php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. GentleChef):
```

```
> Produit
```

```
Your entity already exists! So let's add some new fields!
```

```
New property name (press <return> to stop adding fields):
```

```
> description
```

```
Field type (enter ? to see all types) [string]:
```

```
>
```

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
> yes
```

```
updated: src/Entity/Produit.php
```

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
>
```

La couche métier: Les entités

Insertion des données

```
namespace App\Controller;
use App\Entity\Produit;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\HttpFoundation\Response;
class ProductController extends AbstractController
{
    /**
     * @Route("/productadd/{nomp}", name="productadd")
     */
    public function CreateProduct($nomp):Response
    {
        $entityManager=$this->getDoctrine()->getManager();
        $Produit=new Produit();
        $Produit->setNom($nomp);
        $entityManager->persist($Produit);
        $entityManager->flush();
        return new Response("Produit enregistré");
    }
}
```