
R4.01

Architecture Logicielle

MVC : Le Contrôleur

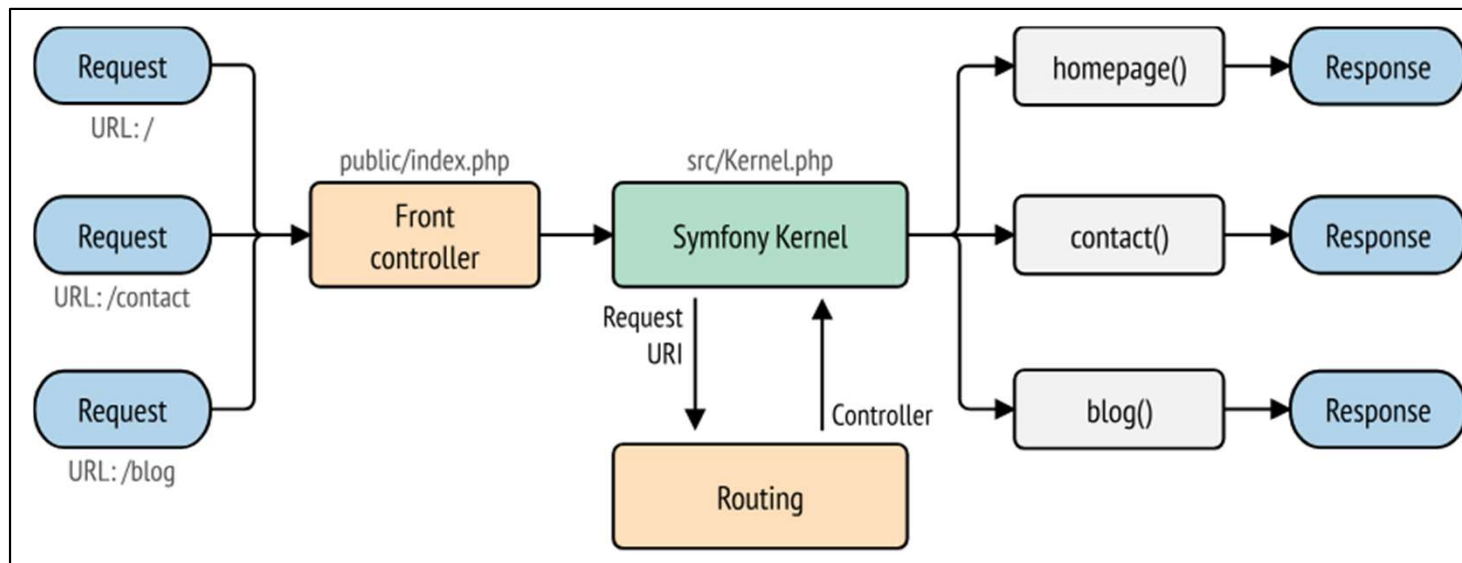
Requête, Session, Erreur 404, Réponse

Contrôleurs Imbriqués

<https://symfony.com/doc/current/controller.html>

Rôle d'un Contrôleur

- Un **Contrôleur** est une méthode à l'intérieur d'une **Classe Contrôleur**.
- Un **Contrôleur** est associé à une **Route** qui définit l'URL qui va déclencher son exécution
- Le rôle d'un **Contrôleur** est donc de traiter une **requête HTTP (classe Request)**, d'exécuter la logique applicative qui lui est assignée, et de retourner une **réponse HTTP (classe Response)** :
 - Une page **HTML** (une vue, produite à partir d'un *template* Twig par exemple)
 - Du **JSON** ou du **XML** (dans le cas d'une API REST)
 - Une **redirection** vers une autre **route**
 - Une erreur **404** si le contrôleur ne peut pas répondre à la requête demandée
 - ...
- **Un contrôleur NE DOIT PAS implémenter de code « métier »**
- **Un contrôleur utilise des Services pour obtenir des données métier et construire sa réponse:**
 - **Services techniques** fournis par Symfony (Twig, Session, Doctrine, Mailer, ...)
 - **Services métier** développés spécifiquement pour l'application (cf TP : **BoutiqueService**, **PanierService**)



Contrôleur – Exemple minimal

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

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

class HelloController {

    #[Route('/hello/{nom}/{prenom}', name: 'app_hello_index')]

    public function index(string $nom, string $prenom) : Response {
        return new Response('<html><body>Hello ' . $prenom . $nom . ' !</body></html>');
    }

}
```

- Le contrôleur **HelloController::index** est associée à la **route** nommée arbitrairement **app_hello_index**
- L'**URL** associée à cette **route** est de la forme **/hello/{nom}/{prenom}**
- Toute **URL** de la forme **/hello/*/ *** envoyée à l'application déclenchera ce contrôleur
- La **route** comporte 2 paramètres appelés **{nom}** et **{prenom}**
- Le **contrôleur** comporte donc lui aussi 2 paramètres **\$nom** et **\$prenom**
- Le **contrôleur** retourne un objet de la classe **Response** qui encapsule la réponse HTTP qui, sur cet exemple, contiendra un peu de HTML

Contrôleur - Paramètres

- Symfony fait correspondre chaque argument du contrôleur avec un paramètre portant le même identifiant dans sa route
- L'ordre des arguments dans le contrôleur n'a pas d'importance
- **Chaque argument déclaré dans le contrôleur doit correspondre à un paramètre de la route**
- **Mais tous les paramètres de la route n'ont pas besoin d'être utilisés par le contrôleur**
- Chaque route possède un paramètre spécial **_route** qui est égal au nom de la route (par exemple: **hello**).
- Toute la **requête HTTP** qui a été envoyée peut être transmise à son contrôleur sous la forme d'un objet de la classe **Request**.
C'est utile, par exemple, pour récupérer les données envoyées par un formulaire :

```
use Symfony\Component\HttpFoundation\Request;
// ...

#[Route(...)]

public function update(Request $request) : Response {

    $form = $this->createForm(...);
    $form->handleRequest($request);
    // ...
}
```

Contrôleur – Classe de Base

- Symfony fournit une classe de base **AbstractController**
- En héritant de cette classe, votre contrôleur pourra utiliser toutes les méthodes (*helpers*) fournies par sa classe mère :
 - `$this->render(...)` : pour « rendre » un *template* Twig
 - `$this->generateUrl(...)` : pour forger une URL à partir d'un nom de route
 - `$this->redirectToRoute(...)` : pour faire une redirection vers une autre route
 - ... et beaucoup d'autres !

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

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

class HelloController extends AbstractController {

    #[Route('/hello/{nom}/{prenom}', name: 'app_hello_index')]

    public function index(string $nom, string $prenom) : Response {
        return new Response('<html><body>Hello ' . $prenom . $nom . '!</body></html>');
    }

}
```

Contrôleur – Tâches Habituelles

- Après avoir exécuté la logique de l'application, un contrôleur finit habituellement par :
 - **Rediriger** vers une autre page
 - **Transmettre** (« Forwarder ») la requête à un autre contrôleur et récupérer sa réponse pour la renvoyer
 - **Rendre** un *template* (afficher une vue)
 - **Produire une erreur 404**

Contrôleur - Redirection

- Pour rediriger l'utilisateur sur une autre page, utilisez la méthode `redirect()` qui reçoit en paramètre **un nom de route** :

```
public function index() : Response
{
    return $this->redirectToRoute('app_default_index');
}
```

- Si la route attend des paramètres, on les lui transmet via un tableau associatif :

```
public function index() : Response
{
    return $this->redirectToRoute('app_hello_index',
        [ 'nom'      => 'Khonnu',
          'prenom' => 'Alain',
        ]);
}
```

Contrôleur - Forward

- On peut, dans un contrôleur, **transmettre** la requête avec la méthode **forward**
- Plutôt que de rediriger le navigateur de l'utilisateur, cette méthode réalise un appel direct à un autre contrôleur (il n'y a pas de nouvelle requête HTTP)
- La méthode **forward** reçoit en paramètre **l'identifiant d'une méthode** dans un contrôleur (chemin dans l'espace des noms PHP)
- La méthode **forward** retourne l'objet **Response** qui est renvoyé par le contrôleur appelé :

```
public function index(string $name) : Response {  
    $response = $this->forward('App\Controller\OtherController::fancy', array(  
        'name' => $name,  
        'color' => 'green'  
    ));  
    // modifiez encore la réponse ou bien retournez-la directement  
    return $response;  
}
```

- Le tableau passé à la méthode précise les arguments transmis au contrôleur
- Comme pour une route, l'ordre des arguments n'a pas d'importance
- Le contrôleur appelé ici attend donc 2 paramètres name et color :

```
public function fancy($name, $color) : Response  
{  
    // ... crée et retourne un objet Response  
}
```


Contrôleur – Rendre un template

- La plupart des contrôleurs vont délivrer un *template* qui est responsable de la génération du HTML (ou autre format).
- La méthode **renderView** rend un *template* auquel on transmet des paramètres via un tableau associatif et retourne son contenu.
- Le contenu du *template* peut être utilisé pour créer un objet **Response** :

```
public function index($nom, $prenom) : Response
{
    $content = $this->renderView('Hello/index.html.twig',
                                array('nom' => $nom, 'prenom' => $prenom));
    return new Response($content);
}
```

- Ceci peut être effectué en une seule étape à l'aide de la méthode **render()** :

```
public function index ($nom, $prenom) : Response
{
    return $this->render('Hello/index.html.twig',
                        array('nom' => $nom, 'prenom' => $prenom));
}
```

Contrôleur – Page d'erreur

- En cas d'erreur (information non trouvée), il faut utiliser correctement le protocole HTTP et retourner une réponse 404.
- Pour cela, il faut lancer une exception produite de classe **createNotFoundException**
- La méthode **createNotFoundException** crée une exception de la classe **NotFoundHttpException** qui, levée par un throw, sera attrapée par le contrôleur frontal qui produira alors une réponse HTTP 404 :

```
public function index() : Response
{
    $product = // essayer de récupérer un objet depuis un service
    if (!$product) {
        throw $this->createNotFoundException('Le produit n\'existe pas');
    }

    return $this->render(...);
}
```

- Un *template* par défaut est affiché lors d'une réponse 404. Ce template peut être redéfini dans **templates/bundles/TwigBundle/Exception/error404.html.twig**

Contrôleur – Session Utilisateur

- Symfony encapsule toutes les informations sur la session de l'utilisateur dans un objet **Session** disponible grâce à la méthode **getSession** du service **RequestStack**
- Stocker et récupérer des informations depuis la session peut être effectué depuis n'importe quel contrôleur (ou service) :

```
use Symfony\Component\HttpFoundation\RequestStack;

public function index(RequestStack $requestStack) : Response {
    // Récupérer la session grâce au service RequestStack
    $session = $requestStack->getSession();

    // Stocker une variable pour une réutilisation lors d'une future requête utilisateur
    $session->set('foo', 'bar');

    // Récupérer la variable dans un autre contrôleur pour une autre requête
    $foo = $session->get('foo');

    // Récupérer la variable avec une valeur par défaut si elle n'existe pas en session
    $filters = $session->get('filters', array());

    // Tester l'existence d'une variable en session
    if ($session->has('foo')) // ...

    // Supprimer une variable en session
    $session->remove('foo');

    // Supprimer toutes les variables de session
    $session->clear();
}
```

Contrôleur – Session – Messages Flash

- On peut stocker de petits messages qui ne seront gardés dans la session de l'utilisateur que jusqu'à la requête suivante.
- C'est utile lors du traitement d'un formulaire : on fait une redirection et on affiche un message spécial lors de la *prochaine* requête.
- Ces types de message sont appelés « **messages flash** ».

```
public function updateAction() : Response
{
    $form = $this->createForm(...);
    $form->handleRequest($this->getRequest());
    if ($form->isSubmitted() && $form->isValid()) {
        // Effectuer le traitement du formulaire
        $this->addFlash('notice', ['Vos changements ont été sauvegardés!']);
        return $this->redirectToRoute('une_route');
    }
    return $this->render(...);
}
```

- Dans le *template* de la prochaine action, on afficherait le(s) **message(s) flash** contenus dans **notice** :

```
{% for flashMessage in app.flashes('notice') %}
    <div class="flash-notice">
        {{ flashMessage }}
    </div>
{% endfor %}
```

Contrôleur – Objet Response

- La seule contrainte d'une action dans un contrôleur est de retourner un objet **Response**.
- La classe **Response** modélise la réponse HTTP (le message texte rempli avec des en-têtes HTTP et du contenu qui est envoyé au client) :

```
use Symfony\Component\HttpFoundation\Response;  
// crée une Réponse avec un code de statut 200 (celui par défaut)  
$response = new Response('Hello '.$name, Response::HTTP_OK);  
  
// crée une réponse JSON avec un code de statut 200  
$response = new Response(json_encode(array('name' => $name)));  
$response->headers->set('Content-Type', 'application/json');
```

- La propriété **headers** est un objet **HeaderBag** (conteneur) avec plusieurs méthodes utiles pour lire et transformer les en-têtes de la réponse.
- Les noms des en-têtes sont normalisés et ainsi, utiliser **Content-Type** est équivalent à **content-type** ou même **content_type**.

Contrôleur – Objet Request

- En plus des paramètres transmis par le routage, le contrôleur a accès à l'objet **Request** quand il étend la classe Controller de base
- Comme l'objet **Response**, les en-têtes de la requête sont stockées dans un objet **HeaderBag** et sont facilement accessibles.

```
$request = $this->getRequest(); // récupérer la requête
                                   // si non demandée dans les paramètres


$request->isXmlHttpRequest();    // est-ce une requête Ajax?

$request->getPreferredLanguage(array('en', 'fr')); // info sur les langues

$request->query->get('page');     // récupérer la valeur d'un paramètre $_GET

$request->request->get('page');   // récupérer la valeur d'un paramètre $_POST
```

Contrôleurs imbriqués (1)

- Dans certains cas, un *template* peut avoir besoin d'inclure un autre *template* dont le contenu doit être « alimenté » par son propre contrôleur
- Exemple : un menu latéral dans le **layout** qui contient les trois produits les plus vendus.
 - La récupération des trois articles les plus récents peut nécessiter une requête vers une base de données et de réaliser d'autres opérations « logiques » qui ne **doivent pas** être effectuées dans un *template*.
- La solution consiste simplement à **imbriquer** les résultats d'un contrôleur dans un *template*.
-  Cette technique est très pratique mais ne respecte pas du tout le pattern MVC !

Contrôleurs imbriqués (2)

- Dans un premier temps, créer un contrôleur qui retourne un certain nombre de produits les plus vendus :

```
// src/Controller/ArticleController.php

class ArticleController extends AbstractController {

    public function plusVendus($max = 3) : Response {

        // Demander au « modèle » les $max articles les plus vendus
        $articles = ...;

        return $this->render('Article/plusVendus.html.twig',
                             array('articles' => $articles));
    }
}
```

 Aucune route ne doit être associée à un contrôleur imbriqué !

- Le template « plusVendus » rendu par ce contrôleur :

```
{# templates/Article/plusVendus.html.twig #}
{% for article in articles %}
    <a href="/article/{{ article.id }}">
        {{ article.intitule }}
    </a>
{% endfor %}
```

 Ne pas faire hériter du *layout* ce template qui va justement être « inséré » dans le *layout* !

Contrôleurs imbriqués (3)

- Pour inclure le résultat de ce contrôleur dans le template **layout** (ou dans n'importe quel autre template), il faut, dans une balise Twig `{{ ... }}` :
 - Appeler le contrôleur imbriqué avec la fonction **controller**, en indiquant le chemin dans l'espace de nom PHP du contrôleur imbriqué (`App\\Controller\\ArticleController::plusVendus`)
 - Insérer le résultat de cet appel (*qui est une réponse HTTP contenant du HTML*) dans le template à l'aide de la fonction **render**

```
{# templates/base.html.twig #}  
...  
<div id="sidebar">  
    {{ render(  
        controller('App\\Controller\\ArticleController::plusVendus',  
                    {'max': 3}  
        )  
    }}  
</div>  
...
```

Si le contrôleur imbriqué attend des paramètres, on passe leurs valeurs dans un tableau associatif Twig : `{ 'nomParam' : valeurParam, ... }`