Web - TP Symfony n^o 2
G. Subrenat, R. Zrour

Licence Informatique L3 - 2022-2023
UFR SFA - Université de Poitiers

Première approche de Symfony

Table des matières

Le code du TP est disponible à : https://gitlab.com/subrenat/l3-web-2022-23
dans la branche *b-TP2* : https://gitlab.com/subrenat/l3-web-2022-23/-/tree/b-TP2

1 Résumé

Dans ce TP nous verrons le principe de fonctionnement de Symfony sur un mini-site, à l'exception des bases de données (qui sont une partie importante) :

- arborescence de Symfony
- création de routes,
- création d'un contrôleur et d'actions,
- création de vues (templates Twig)
- fonctionnement de la toolbar

On insiste sur le fait que ce sera un survol ; certains points seront détaillés dans les TPs suivants.

De fait tout ne sera peut-être pas limpide. Le but est de comprendre les principes et les articulations entre les acteurs (routes, contrôleurs et actions, vues) du framework.

Remarques importantes (y compris pour les TPs ultérieurs) :

- Les TP sont très guidés et on peut en faire une bonne partie sans comprendre : il suffit de taper les commandes fournies (qui ont été testées et fonctionnent).
  Bien entendu ce n'est pas le but ; le but est au contraire de comprendre les commandes que vous lancez.
  N'hésitez pas à regarder la documentation, à poser des questions, à faire des tests, ...
- Si une commande que vous lancez affiche un message d'erreur c'est qu'il y a un problème ! Il faut comprendre ce qu'il se passe et corriger l'erreur ; il est inutile de continuer le TP et de lancer les commandes suivantes.
- Pourquoi les deux remarques précédentes qui semblent triviales ? Parce que c'est du vécu des enseignants et qu'il nous semble nécessaire d'insister.

2 Arborescence de Symfony

Travail à effectuer

Cette section décrit l'arborescence de Symfony.

Vérifiez que la description correspond à ce que vous avez dans votre répertoire. Regardez également si des fichiers sont déjà présents dans les répertoires.

L'arborescence de Symfony comportent plusieurs répertoires qui ont chacun un rôle très précis, rôles qu'il convient de respecter si l'on désire que de nouveaux développeurs puissent intégrer le projet sans problème.

Voici une brève description des répertoires :

bin il contient un script PHP (nommé *console*) à exécuter dans un terminal de commandes. Le mode console est très utilisé avec Symfony : création de contrôleurs, gestion du cache, ...

config il contient tout ce qui commun au site, mais le code source (i.e. votre code) sera ailleurs. Il s'occupe par exemple des chargements des modules, des accès à la base de données, ... Bref il s'occupe de la configuration du site.

migrations
Il contient les schémas successifs de la base de données.

public c'est le point d'entrée du site, il contient tout ce qui doit être directement accessible par l'internaute : images, CSS, JavaScript notamment. Il contient également le fichier d'entrée (contrôleur frontal) : *index.php*.
Ce devrait (doit) être le seul répertoire accessible de l'extérieur : reconfigurer Apache et créer un VirtualHost est une bonne idée.

src il contient le code source du site : contrôleurs et tout ce qui en découle. C'est donc dans ce répertoire que l'on passe le plus clair de son temps.

src/Controller
contient les contrôleurs

src/DataFixtures
: permet d'alimenter la base de données (généralement avec des jeux de tests)

src/Entity
: contient les modèles (base de données)

src/Form
: gestion des formulaires

src/Repository
: scripts d'interrogation de la base de données

templates
: il contient les vues (du Twig pour nous) ; c'est la partie du code du site qui n'est pas dans le répertoire *src*.

translations
: pour les sites multilingues.

var
: c'est un répertoire utilisé en interne par Symfony, notamment pour les logs et le cache. Rappelez-vous qu'il faut parfois le vider (en mode console), et il ne faut rien déposer soi-même dedans.

vendor
: il contient les bibliothèques externes utilisées par le site, par exemple : Doctrine (BD), Symfony [1], Twig, ...

3 La console et le cache

Travail à effectuer

Outre comprendre les explications, vous pouvez lancer les commandes proposées pour voir leurs effets.
N'hésitez pas à tester des variantes.

Beaucoup de manipulations se font en mode console (vider le cache, créer un contrôleur, manipuler la base de données, ...). Dans un second temps on utilise l'éditeur de texte (enfin plutôt un IDE) pour modifier et créer des fichiers.

Même si le premier abord est rebutant, le but de la console est de faciliter la vie du programmeur [2]. L'apprentissage est un peu long, mais le gain est rapidement appréciable [3].

Rappels :

- Toutes les commandes en mode console se font sur le **serveur**. Il est ensuite à la charge du développeur de rapatrier (*download*) les changements sur sa machine de développement (via *git*, ou l'IDE, ou à la main).
- Ensuite les changements de code se font sur la machine locale et doivent être envoyés (*upload*) sur le serveur avant d'utiliser le site dans le navigateur.

Mais si le serveur et la machine de développement sont le même ordinateur (ce qui est vraisemblablement le cas), il n'y a plus à se préoccuper de tout cela.

On se place à la racine du site sur le serveur [4], par exemple dans `~/public_html/tp`, puis on utilise le script PHP *bin/console* (pour vider le cache par exemple).
Lancez la commande sans argument, et vous aurez quelques explications sur son fonctionnement. Il y a deux manières équivalentes de lancer la console :

```
serveur$ php bin/console
```

```
serveur$ symfony console
```

L'affichage est long avec un récapitulatif des actions possibles.

On retrouve par exemple la manipulation du cache et vous pouvez demander à la console une aide :

1. Symfony est une bibliothèque externe au même titre que les autres.
2. Le but des outils fournis par les concepteurs de Symfony est bien entendu de faciliter la vie des développeurs. Ce sera à vous de juger si le but est atteint, et si ce n'est pas le cas qui est fautif.
3. De mon point de vue, le but est atteint.
4. dernière fois que l'on précise que la console s'utilise sur le serveur

```
serveur$ symfony console help cache:clear
```

Lorsqu'on fait des modification sur le code source :

- en mode production, il est fréquent de vider le cache [5] :

```
serveur$ symfony console cache:clear --env=prod
```

- en mode développement, il est aussi parfois nécessaire de le faire [6] :

```
serveur$ php bin/console cache:clear
```

Point de cours

Qu'est-ce qu'un cache ? Comment fonctionne-t-il ? À quoi sert-il ?
Voici des questions incongrues car nous en manipulons quotidiennement. Mais un rafraîchissement de la mémoire [a] peut être utile.

De manière générale, un cache est une zone qui stocke les résultats de calculs, les notions de "calcul" et "résultat" étant très larges.

L'avantage et le but sont la rapidité : lorsqu'on a besoin à nouveau d'un résultat, il n'y a qu'à le récupérer dans le cache évitant ainsi des calculs potentiellement coûteux.
Notons toutefois qu'il y a un surcoût lors du premier accès car il faut remplir ce cache ; donc mettre en cache une donnée qui ne servira plus est inutile et pénalisant, mais ce n'est pas toujours prévisible.

L'inconvénient est double :
On occupe un espace supplémentaire (on n'a pas rien sans rien).
Si les données sources ont changé, le résultat du cache est faux (on parle d'incohérence de cache) ; il faut alors le vider ou le mettre à jour.

Voici quelques exemples de cache :

- cache des fichiers pour la navigation sur internet. Le plus connu est celui de votre navigateur qui stocke, par exemple, les images ou le CSS des pages consultées. Mais il peut également y avoir un cache équivalent sur le pare-feu de votre entreprise. Les serveurs ont également des caches lorsque des clients font des requêtes identiques.
- cache DNS. Toujours dans le domaine d'internet les DNS traduisent une URL en adresse IP. Il y a des "DNS cache" à différents niveaux : pays, entreprise, site, ...
  Comme il y a énormément de modifications, les "DNS cache" doivent se mettre à jour régulièrement. C'est pour cela, lorsque vous changez l'IP de votre site, qu'il faut plusieurs heures pour que les DNS soient au courant.
- cache d'un contrôleur de disque. Une partie des blocs est stockée dans une zone mémoire du contrôleur, pouvant ainsi éviter des accès physiques au disque.
- cache mémoire pour le disque dur. Une partie des fichiers sont stockés en mémoire RAM pour les mêmes raisons.
- mémoire cache d'un processeur. C'est une mémoire ultra-rapide (bien plus que la RAM, et bien plus chère aussi) qui stocke une partie de la RAM : données ou code.
- ...
- cache de frameworks. On peut aussi citer les frameworks de développement qui mettent des données et du code en cache.

Toute la problématique est d'avoir un cache cohérent (i.e. à jour) ; il y a plusieurs stratégies plus ou moins efficaces et correctes :

- durée de vie limitée d'une donnée dans le cache,
- demande systématique de validité d'une donnée dans le cache (en se basant sur la date par exemple),
- invalidation automatique : c'est la source de la donnée qui prévient tous les caches lorsqu'il

5. pratiquement à chaque modification du code du site
6. mais c'est extrêmement rare

y a un changement (difficilement applicable sur internet),
- pas de stratégie : c'est à l'utilisateur de penser à vider le cache régulièrement.

Pour Symfony :
- en mode "prod" : c'est la dernière stratégie qui est retenue. Cela peut sembler curieux mais cela permet au serveur de ne pas passer du temps à tester la validité du cache et ainsi d'être le plus efficace possible. Et on travaille extrêmement rarement en mode "prod", aussi n'est-il pas compliqué de penser à vider le cache explicitement.
- En mode "dev" le cache se met presque toujours à jour automatiquement (dans de rares cas ce n'est pas fait et il faut avoir cet état de fait en mémoire). L'inconvénient est que le site est moins rapide, mais il serait insuportable de devoir vider le cache manuellement à chaque changement du code source.

a. mémoire cache bien sûr !

Nous travaillerons cette année uniquement en mode "dev" ; mais quand une modification du site ne fonctionne pas, pensez à vider le cache avant de chercher une erreur éventuelle.

4 Le routeur et les routes

Travail à effectuer

Cette section est uniquement un point de cours.

Point de cours

Habituellement une URL désigne un fichier (html ou php par exemple) et le chemin qui y mène dans l'arborescence du serveur.

Prenons l'exemple : http://monserveur/users/login.php.
On peut deviner (avec une forte probabilité de réussite) qu'il existe sur le serveur un fichier nommé *login.php* qui se trouve dans le répertoire *users* lui-même à la racine du site (par exemple */var/www*).

Et bien, notamment avec les frameworks, il faut oublier cette logique. Nous manipulerons des URLs qui ressemblent à :
http://monserveur/image/compare/34/78
Il est peu vraisemblable qu'il existe un fichier nommé *78* dans un répertoire nommé *34*. Et même nous verrons que les répertoires *image* et *compare* n'existent pas.
On peut même deviner que l'on va comparer les 34me et 78me images d'une base de données.

Le point d'entrée sur un site géré par un tel framework est une méthode (que l'on appelle *action*) d'un contrôleur (qui est donc une classe dédiée à ce rôle).

Définition : une *route* associe une URL à une action d'un contrôleur, ainsi que ses paramètres.
Dans l'exemple ci-dessus, l'URL pourrait correspondre à la méthode *compare* de la classe *Image* avec les paramètres 34 et 78.

Définition : le *routeur* :
- analyse l'URL envoyée par l'internaute,
- cherche si une route lui correspond.
- s'il y a succès, renvoie la classe, l'action et les paramètres correspondants.

Nous choisirons le mode de configuration dit "annotations" : une route sera décrite dans les attributs juste en dessus de l'action qui lui est liée. Autrement dit les routes seront spécifiées dans les classes contrôleurs.

Remarque : les attributs sont arrivés avec PHP 8. Auparavant les routes (et d'autres informations) étaient décrites dans des commentaires en dessus des actions.
Le principe reste le même, seule la syntaxe change [7].

7. C'est un peu plus subtil : les commentaires permettaient des directives imbriquées, ce qui n'est pour l'instant pas le

Pour utiliser concrètement les routes, il faut d'abord créer un contrôleur et une action, ce qui est l'objet de la section suivante.

5 Le contrôleur

5.1 Un peu de théorie

Travail à effectuer

Juste lire les explications.

Point de cours

Un contrôleur est un chef d'orchestre de requêtes utilisateur. Techniquement un contrôleur est une classe.

Il est le point d'entrée d'une requête et c'est lui qui utilisera les autres composants : base de données (Entités), vues (templates Twig), formulaires, ... pour générer la réponse à l'utilisateur.

C'est lui qui prendra les décisions et orientera le résultat en fonction du contexte : utilisateur connecté ou non, avec quels droits, ...

Un contrôleur peut gérer plusieurs URLs grâce des actions et une action gère une seule [a] URL. Techniquement une action est une méthode du contrôleur.

Un site comporte généralement plusieurs contrôleurs et un contrôleur gère un ensemble cohérent de requêtes utilisateur. Par exemplaire :

- un contrôleur pour gérer l'accueil du site
- un contrôleur pour gérer les utilisateurs (connexion, déconnexion, gestion du profil, ...)
- un contrôleur pour gérer les produits d'un magasin
- ...

a. c'est un peu plus subtil avec les URLs paramétrables

Les contrôleurs sont rangés dans le répertoire *src/Controller* et peuvent être répartis dans des sous-répertoires.
Il faut des sites "conséquents" pour avoir besoin de ranger les contrôleurs dans des sous-répertoires. Pour avoir un ordre d'idée, le site qui va être construit au fil des TPs comportera moins de 10 contrôleurs.

Ceci dit, les exercices vont nécessiter deux types de contrôleurs :

- des contrôleurs de test/entraînement
- des contrôleurs pour le "vrai" site

Afin de ne pas mélanger les deux types de contrôleur, les contrôleurs de test seront dans le sous-répertoire *Sandbox* (i.e. *src/Controller/Sandbox*), et les autres contrôleurs seront directement dans le répertoire dédié (i.e. *src/Controller*).

5.2 Création d'un contrôleur

Travail à effectuer

Nous allons enfin créer des pages !

Il faut taper les commandes proposées pour obtenir un premier contrôleur, puis apporter les modifications demandées.

Rappelons que le but n'est pas de taper les commandes pour vérifier qu'elles fonctionnent (car elles

cas avec les attributs PHP

fonctionnent) mais de les comprendre, de voir le code généré et de visualiser le résultat dans le navigateur.

Il existe une commande console pour générer un contrôleur (mais il ne serait pas très compliqué de l'écrire à la main).
On rappelle que la console se lance sur le serveur à la racine du site.

La commande est "`make:controller`". Commençons par lire l'aide de cette commande :

```
serveur$ php bin/console help make:controller
```

ou encore :

```
serveur$ symfony console help make:controller
```

Nous allons générer un contrôleur sans template (i.e. sans fichier html/twig) pour deux raisons :
- on garde l'étude des templates pour un plus tard dans le document.
- le template généré automatiquement est complexe, et un TP sera consacré à cette complexité.

Généralement les classes contrôleur sont directement dans le répertoire *src/Controller*, c'est à dire qu'il est très rare de créer des sous-répertoires, du moins pour des sites de taille raisonnable [8].
Ceci dit tous les contrôleurs servant à illustrer les explications au fil des sujets de TP seront dans un sous-répertoire nommé *Sandbox*

Dans la commande ci-après, on ne précise pas le nom du contrôleur qui sera alors demandé interactivement. Le nom du contrôleur sera *Overview* dans le répertoire *src/Controller/Sandbox*. La commande est alors (ne pas oublier l'option *--no-template*) :

```
serveur$ symfony console make:controller --no-template

 Choose a name for your controller class (e.g. TinyGnomeController):
 > Sandbox\Overview

 created: src/Controller/Sandbox/OverviewController.php

  Success!

 Next: Open your new controller class and add some pages!
serveur$
```

On remarque :
- *ligne 4* : on précise le nom du répertoire et le nom du contrôleur sans le suffixe *Controller*.
  Important : le nom du répertoire et le nom du contrôleur sont séparés par un backslash et non un slash. Cela n'a aucun rapport avec le séparateur de Windows ou Linux ; en fait il s'agit de *namespaces*.
- *ligne 6* : le suffixe *"Controller"* a été ajouté automatiquement (on aurait aussi pu le préciser explicitement) : ce suffixe est obligatoire pour une classe contrôleur.
- *ligne 6* : le script indique quel(s) fichier(s) a(ont) été créé(s), en l'occurrence un seul dans notre cas : *src/Controller/Sandbox/OverviewController.php*.
- le code généré est visible à :
  https://gitlab.com/subrenat/l3-web-2022-23/-/blob/f2ca29222144e169686ebfdb2159d6757ad07d90/src/Controller/Sandbox/OverviewController.php

Nous allons légèrement modifier ce fichier avant de tester le site :
- Renommez le nom de l'action (i.e. la méthode) en *"indexAction"* : c'est une convention de codage [9] de suffixer le nom des méthodes reliées à une route par *"Action"*.
- Remplacez le code de l'action par la ligne :
  `return new Response('Hello World!');`
  L'IDE vous propose plusieurs type *Response* ; choisissez :
  `Symfony\Component\HttpFoundation\Response`
  le *use* correspondant est automatiquement inséré en tête de fichier.

8. bis repetita
9. qui n'est pas obligatoire. J'aime bien, juste en regardant le nom d'une méthode, savoir si c'est une action (i.e. une méthode liée à une URL) ou une méthode interne.

- Le type de retour de la méthode n'est alors plus valide : ce doit être *Response* et non *JsonResponse*. L'IDE *PhpStorm* le signale et propose de faire la correction automatiquement.
- Le *use* sur le type *JsonResponse* n'a plus lieu d'être et peut être supprimé.
- Enfin, dans l'annotation en dessus de la méthode, dans l'attribut *name* enlevez le préfixe *"app_"*.

Le nouveau code de la classe est accessible à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/5634398c406564edb5c2b1367a8029663e09493b/src/Controller/Sandbox/OverviewController.php

En voici une copie ici :

Sandbox/OverviewController.php

```
1  <?php
2
3  namespace App\Controller\Sandbox;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\HttpFoundation\Response;
7  use Symfony\Component\Routing\Annotation\Route;
8
9  class OverviewController extends AbstractController
10 {
11     #[Route('/sandbox/overview', name: 'sandbox_overview')]
12     public function indexAction(): Response
13     {
14         return new Response('Hello World!');
15     }
16 }
```

Voici un descriptif :

- *ligne 3* : le nom du namespace. Toutes les classes de Symfony sont dans des namespaces nommés selon le chemin à partir du répertoire *src* (comme les packages de Java), avec le préfixe "`App\`" en plus.
- *lignes 5 à 7* : les classes extérieures utilisées (comme les *import* de Java). Notamment à ligne 6 il y a la nouvelle classe utilisée.
- *ligne 9* : nom de la classe avec son héritage.
- *lignes 11 à 15* : une action de la classe (route et code)
- *ligne 11* : description de la route. Le premier paramètre est l'URL, le second son nom interne (dont nous verrons l'utilisation ultérieurement).
- *ligne 12* : prototype de l'action. Pour l'instant elle n'a pas de paramètre et le type de retour est la classe *Response* (cf. ligne 6).
- *ligne 14* : construction d'une instance de la classe *Response*. Le paramètre est une chaîne de caractères qui est le code HTML envoyé au navigateur.

Note importante : nous mettons ici le code HTML directement dans le contrôleur, c'est une violation du modèle MVC et donc à proscrire absolument (sauf si on vous donne l'autorisation expresse).
Et le code est incomplet, il manque le *doctype*, les balises *head* et *body*, ...

Il reste à voir le résultat, ce que nous allons faire tout de suite.

5.3 Test de la première action du contrôleur

Travail à effectuer

Il faut uniquement afficher la nouvelle page dans le navigateur.

Nous pouvons enfin afficher notre première page dans le navigateur. D'après le code de l'action, il suffit de rajouter *"/sandbox/overview"* après l'URL qui désigne la racine de votre site dans la barre d'adresse du navigateur.
Si on utilise Apache, on a par exemple :
http://localhost/L3/tp/public/sandbox/overview
Si on utiliser le serveur de Symfony, on a vraisemblablement :
http://localhost:8000/sandbox/overview

Rappel : si il y a une erreur, commencez à vider le cache et recharger la page avant de chercher une erreur dans votre code.

5.4 Où est passée la toolbar ?

Travail à effectuer

Il faut apporter une modification dans le code de l'action.

Vous avez peut-être noté que la toolbar a disparu alors que nous sommes en mode développement, ce qui n'est pas normal et très handicapant pour débuguer.

La toolbar est une barre d'outils (!) présente en bas de la fenêtre du navigateur et permet, entre beaucoup d'autres choses, de débuguer et d'analyser les requêtes.

Revenez sur la page d'accueil du site (http://localhost:8000) pour la visualiser :

Voir la section 7 pour plus de détails.

Retournez sur la page générée par le nouveau contrôleur.

Le code gérant la toolbar est automatiquement inséré, par le framework, avant la balise *</body>*. Or le code de la page ne contient pas cette balise fermante.
Vérifiez le en affichant le code source de la page (*Ctrl-u* sous Firefox).
Le code est particulièrement succinct [10] et reflète exactement le contenu de la vue (paramètre du constructeur de *Response*).

Éditez le code de l'action et encapsulez le texte entre les balises *<body>* et *</body>*.
Le nouveau code de la classe est accessible à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/509a1401066a21412ec370ac9e19cc28c44fcf6a/src/Controller/Sandbox/OverviewController.php

Et voici la ligne modifiée :

Sandbox/OverviewController.php

```
14        return new Response('<body>Hello World!</body>');
```

Rafraîchissez la page : la toolbar est revenue. Consultez le code source de la page : Symfony a rajouté une (longue) ligne pour la toolbar.

On peut se demander pourquoi les développeurs de Symfony n'ont pas rajouté le code la toolbar dans les vues systématiquement.
On pourrait citer Philippe Destouches : "la critique est aisée, mais l'art est difficile" [11]. Mais ce n'est pas une raison pour ne pas faire de critiques, surtout si elles sont constructives.

Mais en l'occurrence la critique est infondée.

- une page HTML correcte contient systématiquement la balise "*body*" et donc Symfony peut raisonnablement faire l'hypothèse que cette balise est présente.
- et surtout une vue ne contient pas forcément du HTML, ce pourrait être du JSON, du PDF, ... Et dans de tels cas, la présence d'une toolbar serait pour le moins inadéquate.

6 Les template Twig

6.1 Un peu de théorie

Travail à effectuer

Il y a juste à lire le point de cours.

Point de cours

Rappel : il n'est pas souhaitable [a] de mettre le code des vues directement dans les contrôleurs :

- cela viole le principe du modèle MVC,

10. et syntaxiquement incorrect : la structure d'un document HTML n'est pas respectée.
11. "certes, mais sans liberté de blâmer, il n'est pas d'éloges flatteurs" écrivait Beaumarchais (merci Wiktionnaire).

- les développeurs qui codent les vues n'ont pas à connaître Symfony (même si cela est a priori surprenant).

Les vues doivent donc être dans des fichiers dédiés.

Définition : un *template* est un fichier affichant une page HTML dynamique sans PHP.

Twig est un de ces langages qui se veut plus abordable et lisible que PHP toujours dans l'optique que le développeur des vues n'est pas obligatoirement un informaticien chevronné.

Voici quelques exemples de Twig :

`{{ titre }}`	affiche le contenu de la variable *titre*
`{{ titre|upper }}`	affiche le contenu de la variable *titre* en majuscules
`{% for elt in ensemble %}`	fait une boucle sur l'objet itérable *ensemble*

Twig possède de nombreuses fonctionnalités que nous n'abordons pas pour l'instant.

Notons qu'utiliser Twig n'est pas une obligation : il est possible d'écrire les vues directement en PHP.

a. c'est un euphémisme : c'est tout simplement proscrit.

6.2 Création d'une seconde action

Travail à effectuer

Le but est, pour générer une seconde page, d'écrire une seconde action en suivant les indications.

Créez une seconde action que l'on pourra nommer *hello2* (ou plus exactement *hello2Action*). Pensez à changer l'URL (*"/sandbox/overview/hello2"*) et le nom interne (*"sandbox_overview_hello2"*).

Le nouveau code de la classe est accessible à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8bda0f33a4442cbf9ece42fec5d8566133a351f4/src/Controller/Sandbox/OverviewController.php

Et voici la nouvelle action :

Sandbox/OverviewController.php

```
17     #[Route('/sandbox/overview/hello2', name: 'sandbox_overview_hello2')]
18     public function hello2Action(): Response
19     {
20         return new Response('<body>Hello World number 2!</body>');
21     }
```

En l'occurrence dans ce contrôleur nous faisons le choix que toutes les routes commencent par *"/sandbox/overview"*.

Un nom interne (attribut *name*) doit être unique dans tout le site et non pas uniquement au sein du contrôleur, aussi faut-il adopter une stratégie de nommage. Habituellement on adopte :

`<nom complet contrôleur>_<nom action>`

Soit dans notre cas :

`sandbox_overview_hello2`

On note que :

- on a choisi, arbitrairement, la notation snake case (c'est la convention utilisée par Symfony)
- on omet les suffixes *Controller* et *Action*

Vérifiez que la nouvelle route est correcte, et que l'ancienne fonctionne toujours [12].

6.3 Principe des fichiers template

12. test de non régression

Travail à effectuer

Il y a juste à lire les explications.

Une vue (ou un template) est associée à une action d'un contrôleur. Une vue est un fichier contenant le code (HTML et Twig pour nous) générant du HTML pour le navigateur.

Les vues se mettent dans le répertoire *templates* du site, ou dans des sous-répertoires.

Voici la règle de nommage des templates Twig :

- on crée un répertoire (dans *templates* donc) par contrôleur
- si le contrôleur est dans un sous-répertoire de *src/Controller* alors on crée un sous-répertoire de même nom dans *templates*.
  Dans notre cas il faudra créer le répertoire *templates/Sandbox*.
- Le nom du répertoire est celui du contrôleur sans le suffixe (dans notre cas : *templates/Sandbox/Overview*).
- le fichier vue a le même nom que l'action sans le suffixe *"Action"* et avec l'extension *".html.twig"* (dans notre cas : *templates/Sandbox/Overview/hello2.html.twig*).

Pour l'instant ne créez pas ce fichier.

6.4 Appel du template dans le contrôleur

Travail à effectuer

La but, en suivant les indications, est d'appeler le code du template à partir de l'action.
Mais on ne crée pas encore le fichier template, ce qui déclenchera une erreur.

Nous allons modifier le corps de la méthode *hello2Action* du contrôleur pour lui indiquer de récupérer notre vue, ceci à la place du code basique actuellement présent.

Toute une série de méthodes permettent de manipuler les templates et sont présentes dans la classe *AbstractController*[13]. C'est pourquoi notre classe hérite de *AbstractController*.

Normalement le code d'une action se passe en deux temps :

- on demande au template de s'exécuter et de renvoyer le code HTML généré.
- on construit une *Response* avec le texte récupéré.

Mais la classe *AbstractController* fournit la méthode *render* qui effectue ces deux opérations. Le code de l'action est alors :

Sandbox/OverviewController.php

```
17      #[Route('/sandbox/overview/hello2', name: 'sandbox_overview_hello2')]
18      public function hello2Action(): Response
19      {
20          return $this->render('Sandbox/Overview/hello2.html.twig');
21      }
```

Le code complet de la classe est accessible à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/09598a655c89cee0a73c455210264825a99eb512/src/Controller/Sandbox/OverviewController.php

Essayez de charger cette nouvelle URL (i.e. http://localhost:8000/sandbox/overview/hello2) dans le navigateur et on tombe sur une erreur, ce qui normal puisque le fichier *twig* n'existe pas encore. Le descriptif de l'erreur est particulièrement précis et clair :

- en haut de page, le libellé de l'erreur

13. qui contient bien d'autres méthodes

Le message indique bien le fichier qu'il cherche
- en milieu de page la ligne de code incriminée

6.5 Création du template

Travail à effectuer

La but est de créer le fichier template cité ci-dessus en suivant les indications.

Créez le fichier Twig en mettant, par exemple, le code suivant à l'intérieur.
Le code de la vue est téléchargeable à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/222b59174bf78ed7536da27b205dcd524dd08916/templates/Sandbox/Overview/hello2.html.twig

templates/Sandbox/Overview/hello2.html.twig

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="UTF-8">
5          <title>Bienvenue !</title>
6      </head>
7      <body>
8          <h1>Bienvenue dans le bac àsable</h1>
9          <p>
10             Si vous voyez ce message, cela signifie que votre premier template Twig (sans code Twig)
11             fonctionne.<br />
12             Et bien sûr : <q>Hello World 2!</q>
13         </p>
14     </body>
15 </html>
```

Et vérifiez que la page se charge correctement.

6.6 Un peu de Twig

Nous parlons de Twig depuis le début de la section sans en avoir fait une seule ligne.

Travail à effectuer

En suivant les explications il s'agit de faire un nouveau couple action/template.
Mais dans cet exercice l'action transmet des informations au template qui les affiche avec du code Twig.

Dans un premier temps, créez une troisième action : *hello3Action* avec sa propre vue, et sans oublier de lui attribuer une route.

Puis on passe deux arguments à la vue, une chaîne et un tableau de chaînes, selon le modèle suivant :
Le code du contrôleur est téléchargeable à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/f398ec7e0883d59be28bfb2ab1857263b0061733/src/Controller/Sandbox/OverviewController.php

src/Controller/Sandbox/OverviewController.php

```
23     #[Route('/sandbox/overview/hello3', name: 'sandbox_overview_hello3')]
24     public function hello3Action(): Response
25     {
```

```php
26        $args = array(
27            'prenom' => 'Gilles',
28            'jeux' => ['A Plague Tale : Innocence', 'WoW', "Mass Effect", 'Life is Strange'],
29        );
30        return $this->render('Sandbox/Overview/hello3.html.twig', $args);
31    }
```

Explications : ce n'est pas le tableau *$args* qui est passé en paramètre à la vue [14] mais ses cases. Autrement dit la vue ne reçoit pas un tableau mais deux variables indépendantes nommées dans notre cas *prenom* et *jeux*.

Dans la vue, placez un code qui ressemble à l'exemple ci-dessous.
Le code de la vue est téléchargeable à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/f398ec7e0883d59be28bfb2ab1857263b0061733/templates/Sandbox/Overview/hello3.html.twig

templates/Sandbox/Overview/hello3.html.twig

```twig
7     <body>
8         <h1>Bienvenue {{ prenom }}</h1>
9         <p>
10            Mes jeux préférés sont :
11            {% set i = 1 %}
12            {% for jeu in jeux %}
13                {{ i }}:{{ jeu }}          {# i peut être avantageusement remplacé par loop.index #}
14                {% set i = i+1 %}
15            {% else %}                     {# utilisé si aucune itération n'est effectuée #}
16                je ne préfère aucun jeu !
17            {% endfor %}
18        </p>
19    </body>
```

Explications :

- *ligne 8* : la variable *prenom* a été transmise par le contrôleur (première case du tableau *$args*). Pour l'afficher, on l'encadre entre {{ et }}.
- *ligne 11* : création et initialisation d'une variable locale au script.
- *ligne 12* : *jeux* est la seconde variable transmise par le contrôleur. *jeu* est une variable locale qui prend successivement, comme valeurs, les contenus des cases du tableau *jeux*.
- *ligne 15* : la clause *else* d'un *for* est exécutée uniquement si aucune itération n'est effectuée, dans notre cas si le tableau *jeux* est vide.

Travail à effectuer

En vous inspirant du code précédent et en suivant les explications ci-dessous, faites un autre couple action/template.

Enfin faites une action *hello4Action* qui a exactement le même code que *hello3* si ce n'est qu'elle se réfère à la vue *hello4.html.twig*.
La vue *hello4* affiche les mêmes informations que *hello3*, si ce n'est que la liste des jeux est dans un tableau HTML : la première colonne contient le numéro du jeu, et la seconde le nom.

Un exemple de solution se trouve à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/26e07684df1d82fe9f1806015a88cd52f4269daf/src/Controller/Sandbox/OverviewController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/26e07684df1d82fe9f1806015a88cd52f4269daf/templates/Sandbox/Overview/hello4.html.twig

7 Aperçu de la toolbar

Travail à effectuer

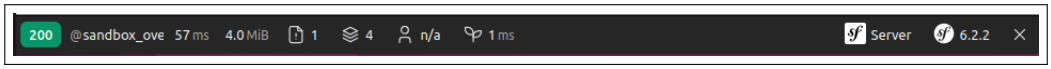
Il s'agit d'explorer quelques aspects de la toolbar.

La toolbar est un outil très puissant pour débuguer et mettre au point un site.

14. Pour l'instant "template" et vue "sont synonymes.

7.1 Survol de la toolbar à la souris

Connectez-vous sur la route de l'action *hello4* du contrôleur *Overview*.



Il y a 9 items sur la toolbar, avec en plus la croix de fermeture. Sur chacun, un popup apparaît au survol de la souris.

Voici la liste de gauche à droite :

- *Request* : quelques informations sur le contrôleur et la route.
- *Performance temps* : temps d'exécution de la requête.
- *Performance mémoire* : occupation mémoire de la requête.
- *Log* : nombre d'erreurs, warnings.
- *Cache* : temps d'accès, nombre d'utilisations. Si on vide le cache et qu'on recharge la page, on a des valeurs différentes.
- *Sécurité*
- *Twig (i.e. vues)*
- *Server* (à droite) : item présent uniquement si c'est le serveur de Symfony qui tourne.
- *Configuration* (à droite) : version de Symfony par exemple.

On peut également cliquer sur un item pour un compte-rendu détaillé.

Cliquez par exemple sur le 2me (ou 3me) item "Performance". On a accès à un graphe très détaillé du temps passé dans chaque module.

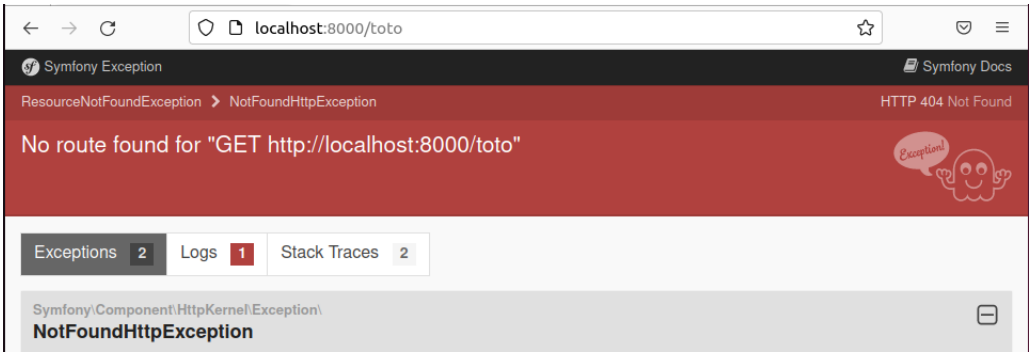
Si on clique sur le premier item, on affiche le tableau de bord de l'outil.

7.2 Routes testées par Symfony

Une utilisation très fréquente de la toolbar est lorsqu'une route n'est pas trouvée alors qu'on est persuadé d'avoir tout codé correctement [15].

Chargez une URL incorrecte, par exemple /toto.

Vous avez à l'écran le message : "No route found for "GET ..." :



Et la toolbar vous indique l'erreur 404 :

Cliquez sur n'importe quel item, puis sur Routing dans le menu à gauche.

On voit l'ensemble des routes que Symfony a testées, et dans quel ordre, sans en trouver aucune. Et on retrouve bien nos 4 routes en fin de liste.

15. ce qui généralement est utopique.

8 La suite

Travail à effectuer

Rien à faire : on expose juste le site qui sera développé dans les prochains TPs.

Le fil conducteur sera la gestion et la distribution de produits manufacturiers :

- caractéristiques des produits
- création, lecture, mise à jour, suppression (CRUD) des produits
- associer un manuel technique à un produit
- associer une ou plusieurs images à un produit
- indiquer dans quels pays ils sont vendus
- indiquer les prix unitaires et les stocks pour chaque magasin
- gestion des utilisateurs

9 Exercice

Travail à effectuer

Créez un contrôleur *Accueil* avec une action *index*[a] branchée sur l'URL "/", autrement dit la racine du site. La vue associée (fichier Twig) affiche un message de bienvenue.

a. Rappel : le nom complet de l'action est *indexAccueil*

Un exemple de solution se trouve à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/45614c9179edfcf0264752788cc4be19e4061af2/src/Controller/AccueilController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/45614c9179edfcf0264752788cc4be19e4061af2/templates/Accueil/index.html.twig

Vous pouvez créer, pour clôturer le TP, une nouvelle version sous *git* (en étant dans le répertoire racine de l'arborescence) :

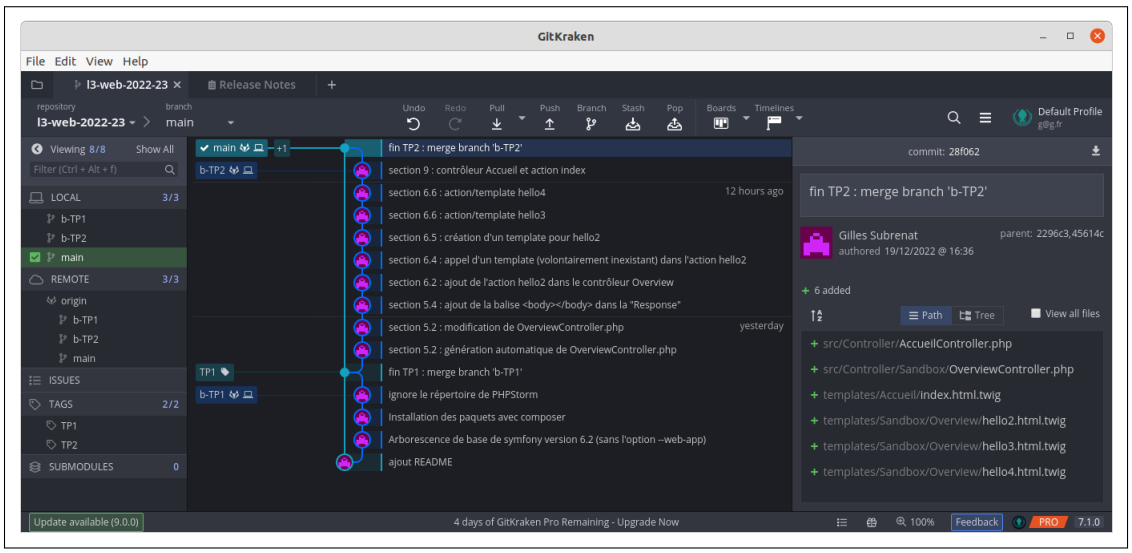
```
serveur$ git add .
serveur$ git commit -m "fin TP2"
[...]
# s'il y a un repository distant
serveur$ git push
[...]
```

10 Git : code source

Le code source à l'issue de ce TP peut-être consulté à :
https://gitlab.com/subrenat/l3-web-2022-23/-/blob/TP2

Les différentes étapes du TP sont dans la branche *b-TP2*.

Voici un screenshot fait avec git-kraken (https://www.gitkraken.com) :


GitKraken
File Edit View Help
l3-web-2022-23
Release Notes
repository
l3-web-2022-23
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Boards
Timelines
Default Profile
Viewing 8/8
Show All
Filter (Ctrl + Alt + f)
LOCAL 3/3
b-TP1
b-TP2
main
REMOTE 3/3
origin
b-TP1
b-TP2
main
ISSUES
TAGS 2/2
TP1
TP2
SUBMODULES 0
main +1
b-TP2
fin TP2 : merge branch 'b-TP2'
section 9 : contrôleur Accueil et action index
section 6.6 : action/template hello4
12 hours ago
section 6.6 : action/template hello3
section 6.5 : création d'un template pour hello2
section 6.4 : appel d'un template (volontairement inexistant) dans l'action hello2
section 6.2 : ajout de l'action hello2 dans le contrôleur Overview
section 5.4 : ajout de la balise <body></body> dans la "Response"
section 5.2 : modification de OverviewController.php
yesterday
section 5.2 : génération automatique de OverviewController.php
TP1
fin TP1 : merge branch 'b-TP1'
b-TP1
ignore le répertoire de PHPStorm
Installation des paquets avec composer
Arborescence de base de symfony version 6.2 (sans l'option --web-app)
ajout README
commit: 28f062
fin TP2 : merge branch 'b-TP2'
Gilles Subrenat
authored 19/12/2022 @ 16:36
parent: 2296c3,45614c
+ 6 added
Path
Tree
View all files
+ src/Controller/AccueilController.php
+ src/Controller/Sandbox/OverviewController.php
+ templates/Accueil/index.html.twig
+ templates/Sandbox/Overview/hello2.html.twig
+ templates/Sandbox/Overview/hello3.html.twig
+ templates/Sandbox/Overview/hello4.html.twig
Update available (9.0.0)
4 days of GitKraken Pro Remaining - Upgrade Now
100%
Feedback
PRO
7.1.0