Web - TP Symfony n^o 3
G. Subrenat, R. Zrour

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

Les routes

Table des matières

1 Résumé **3**

2 Cours **3**

3 Principes des routes : préfixes, variables, ... **4**
3.1 Copie du contrôleur *Overview* . . . 4
3.2 Contrôleur *Sandbox* et préfixes . . . 5
3.3 Routes avec variables . . . 6
3.3.1 Création d'un nouveau contrôleur . . . 6
3.3.2 Action avec une variable simple . . . 7
3.4 Routes avec variables et valeurs par défaut . . . 8
3.5 Routes avec contraintes . . . 10
3.6 exercice *test1* . . . 11
3.7 exercice *test2* . . . 12
3.8 exercice *test3* . . . 13
3.9 exercice *test4* . . . 14
3.10 exercice *test4* encore . . . 14

4 Paramètres injectés **15**
4.1 Nouveau contrôleur . . . 15
4.2 Objet Request . . . 16
4.3 Objet Session . . . 17

5 Déclencher une erreur 404 **18**

6 Les redirections **19**
6.1 Redirection simple . . . 19
6.2 Redirection avec paramètres . . . 20
6.3 Interception des redirections . . . 21

7 Messages flash **22**
7.1 Création des messages flash . . . 23
7.2 Affichage des messages flash . . . 24

8 Application "vente en ligne" **25**
8.1 Base de données . . . 25
8.2 CRUD sur la table produit . . . 26

8.3 CRUD et les tables principales 28
8.4 CRUD et jointures 28
8.5 Autres actions 28

9 Git : code source **29**

Le code du TP est disponible à : https://gitlab.com/subrenat/l3-web-2022-23
dans la branche *b-TP3* : https://gitlab.com/subrenat/l3-web-2022-23/-/tree/b-TP3

1 Résumé

En partant d'une base de données fournie, il s'agit de construire l'ensemble des routes utilisées dans le site.
On ne demande que les routes, et non pas l'accès à la base de données même si cette dernière est fournie pour illustration.

Dans un premier temps des tests sont faits dans les contrôleurs d'entraînement du répertoire *Sandbox*, puis l'apprentissage est appliquée à la "vraie application".

2 Cours

Travail à effectuer

Lire et comprendre les points de cours.

Point de cours

Une route désigne la partie de l'URL située après le chemin qui mène au répertoire *public* d'une arborescence Symfony.

Sous Apache, si :

- le serveur s'appelle *monserveur*
- à la racine du serveur (par exemple */var/www/html*) il y a un répertoire nommée *L3*
- le site symfony est déployé dans le répertoire *tp* lui-même dans le répertoire *L3*

alors l'URL complète est :

http://monserveur/L3/tp/public/image/compare/34/87

et la route [a] est :

/image/compare/34/87

Indépendamment de l'endroit où est déployé le site, si on a lancé un serveur directement dans le répertoire *public* (avec un virtualhost, un *"php -S localhost :8000"* ou un *"symfony server :start"*), l'URL est :

http://localhost:8000/image/compare/34/87

et la route [b] est :

/image/compare/34/87

La route est exactement la même. C'est un point important : les routes dans Symfony [c] sont indépendantes du répertoire d'installation du site [d].

a. abus de langage, il faudrait plutôt parler d'URL locale ; une route contient, outre l'URL locale, d'autres données.
b. cf. footnote ci-dessus
c. et dans les autres frameworks
d. fort heureusement d'ailleurs

Point de cours

Une route dans Symfony n'est pas un chemin vers un fichier, mais est rattachée à une méthode (une action) d'une classe (un contrôleur).

Reprenons la route : */image/compare/34/87*

Il est inutile de chercher un répertoire *"image/compare/34"* et un fichier *"87"* ; ils n'existent vraisemblablement pas.

En revanche on peut "intuiter" qu'il existe une classe *ImageController* avec une méthode *compareAction* qui prend deux entiers en paramètres (qui pourraient être les clés primaires de deux images dans une table de la base de données). On peut supposer également que cette action va indiquer si les deux images sont les mêmes (ou se ressemblent).

Bref les routes sont construites pour donner un sens et non pas pour coller à une arborescence de

fichiers.

On pourrait tout fait avoir la route :
/toto/foo/41/bar/azerty
qui serait liée à l'action *copyright* de la classe *Notes*.
Mais bien entendu ce serait une très mauvaise idée [a] de nommer une telle route ainsi (la route *"/notes/copyright"* est bien plus adéquate).

a. ce qui est un euphémisme : c'est tout simplement à proscrire

Point de cours

Une route est toujours associée à un nom interne qui est très important : dans une application Symfony une route est toujours manipulée par son nom interne, et jamais par l'URL.

Un nom interne doit être unique dans toute l'application. L'habitude est de nommer une route par le couple (nom complet contrôleur, nom action) auquel elle est liée.

Par exemple, si la classe est *ImageController* et l'action est *compareAction*, il est habituel que :
- la route soit */image/compare*
- le nom interne soit *image_compare*

et en conséquence que :
- le répertoire *templates* contienne le sous-répertoire *Image*
- le répertoire *Image* contienne le fichier-vue *compare.html.twig*

Si le contrôleur est dans un sous-répertoire de *Controller*, la route et le nom interne sont rallongés d'autant.

3 Principes des routes : préfixes, variables, ...

Dans un premier temps, vous allez travailler dans les contrôleurs *Sandbox* pour faire quelques tests. On parle de contrôleurs au pluriel car, pour éviter un contrôleur trop volumineux, nous allons le découper en plusieurs (sous-)contrôleurs.

Pour chaque route demandée, vous écrirez une action dans le contrôleur et une vue simplifiées (soit une *Response*, soit un Twig). L'ensemble se contente d'afficher les paramètres de l'action correspondante.

3.1 Copie du contrôleur *Overview*

Travail à effectuer

En suivant les indications, le but est de faire une copie du contrôleur *Overview*.

Afin de garder une trace (sans devoir jouer avec les versions de *git*) du contrôleur *Overview*, il s'agit d'en faire une duplication complète : contrôleur et vues.

Le nouveau contrôleur va s'appeler *Prefix* (ou plus exactement *PrefixController*, mais ce suffixe sera désormais sous-entendu). Ce contrôleur doit se trouver dans le répertoire *src/Controller/Sandbox*.
Et de fait les vues correspondantes doivent se trouver dans le répertoire *templates/Sandbox/Prefix*.

Enfin les routes doivent toutes commencer par */sandbox/prefix* au lieu de */sandbox/overview*. Il faut également penser à changer les noms internes.

Faîtes tous les changements nécessaires et vérifiez que toutes les nouvelles routes fonctionnent.

Le code résultant peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/b0e31af6121ccb3181aaa7416eebe63aa2717022/src/Controller/Sandbox/PrefixController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/b0e31af6121ccb3181aaa7416eebe63aa2717022/templates/Sandbox/Prefix/hello2.html.twig

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/b0e31af6121ccb3181aaa7416eebe63aa2717022/templates/Sandbox/Prefix/hello3.html.twig

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/b0e31af6121ccb3181aaa7416eebe63aa2717022/templates/Sandbox/Prefix/hello4.html.twig

Si vous avez suivi les conventions de nommage exposées dans la section précédente, vous devriez avoir exactement la même chose.

La démarche que l'on pouvait suivre était la suivante :

- recopier le fichier *OverviewController.php* dans *PrefixController.php*
- dans le nouveau fichier, remplacer (en respectant la casse) *overview* par *prefix*, soit 12 remplacements (le nom de la classe, 4 URLs, 4 noms internes et 3 noms de fichiers Twig)
- dupliquer le répertoire *templates/Sandbox/Overview* dans *templates/Sandbox/Prefix* sans changer le contenu des 3 fichiers Twig (sauf si vous voulez les personnaliser)

Voici un extrait du nouveau contrôleur :

Sandbox/PrefixController.php (extrait)

```
3  namespace App\Controller\Sandbox;
```

```
9  class PrefixController extends AbstractController
10 {
11     #[Route('/sandbox/prefix', name: 'sandbox_prefix')]
12     public function indexAction(): Response
13     {
14         return new Response('<body>(Prefix) Hello World!</body>');
15     }
16
17     #[Route('/sandbox/prefix/hello2', name: 'sandbox_prefix_hello2')]
18     public function hello2Action(): Response
19     {
20         return $this->render('Sandbox/Prefix/hello2.html.twig');
21     }
```

3.2 Contrôleur *Sandbox* et préfixes

Pour qu'il n'y ait pas de conflit avec les autres contrôleurs, toutes les routes (i.e. les URLs et les noms internes) du contrôleur *Sandbox/Prefix* doivent être préfixées par *"/sandbox/prefix"* pour les URLs et par *"sandbox_prefix"* pour les noms internes.

La première solution est celle adoptée jusqu'à présent : pour chaque action on précise explicitement que l'URL commence par *"/sandbox/prefix"* et l'attribut *name* par *"sandbox_prefix"*.

Mais il y a de la duplication de code (si on veut changer le préfixe il faut changer toutes les routes) et les attributs sont rapidement très longs.
Aussi est-il possible, et conseillé, de mettre un attribut avant la classe pour imposer les préfixes à toutes les actions.

Travail à effectuer

En suivant les instructions, mettez des préfixes pour le contrôleur *Prefix*.

La syntaxe de l'attribut *Route* à mettre avant la classe est exactement la même que pour les actions :

```
#[Route('<préfixe url>', name: '<préfixe nom interne>')]
```

Ensuite les préfixes précisés en début de classe doivent être supprimés des attributs des routes de toutes les actions (4 actions dans notre cas).

Modifier le contrôleur *Prefix* pour préciser les préfixes en début de classe :

- *"/sandbox/prefix"* pour les URLs
- *"sandbox_prefix"* pour les noms internes

Le code résultant peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/204dcd1c4c19dc3d44847929c98533f6056882f4/src/Controller/Sandbox/PrefixController.php

Voici les parties modifiées du contrôleur :

Sandbox/PrefixController.php (extrait)

```
9  #[Route('/sandbox/prefix', name: 'sandbox_prefix')]
10 class PrefixController extends AbstractController
```

```
11 {
12     #[Route('', name: '')]
13     public function indexAction(): Response
17
18     #[Route('/hello2', name: '_hello2')]
19     public function hello2Action(): Response
23
24     #[Route('/hello3', name: '_hello3')]
25     public function hello3Action(): Response
33
34     #[Route('/hello4', name: '_hello4')]
35     public function hello4Action(): Response
```

Remarques :

- Dans le préfixe des URLs, il n'y a pas de "/" en fin, et donc dans les actions il faudra systématiquement commencer les URLs par un "/". On voit la raison de ce choix dans la route de l'action *index* : l'URL de cette action est le préfixe lui-même, et on veut pas que cette URL se termine par un "/".
- Symétriquement dans le préfixe des noms internes il n'y a pas de "_" en fin, et donc dans les actions il faudra systématiquement commencer les *name* par un "_". La raison est la même que pour les URLs.

Travail à effectuer

Tapez les commandes qui suivent pour vérifier que toutes vos routes sont connues par Symfony.

Pour avoir la liste des routes d'une application, la commande est la suivante :

```
serveur$ php bin/console debug:router --show-controllers
```

ou

```
serveur$ symfony console debug:router --show-controllers
```

Dans l'état actuel de l'application, cela donne :

```
serveur$ symfony console debug:router --show-controllers
 ------------------------- -------- -------- ------ ---------------------------------- ------------------------------------------------------------
  Name                      Method   Scheme   Host   Path                               Controller
 ------------------------- -------- -------- ------ ---------------------------------- ------------------------------------------------------------
  _preview_error            ANY      ANY      ANY    /_error/{code}.{_format}           error_controller::preview()
  _wdt                      ANY      ANY      ANY    /_wdt/{token}                      web_profiler.controller.profiler::toolbarAction()
  _profiler_home            ANY      ANY      ANY    /_profiler/                        web_profiler.controller.profiler::homeAction()
  _profiler_search          ANY      ANY      ANY    /_profiler/search                  web_profiler.controller.profiler::searchAction()
  _profiler_search_bar      ANY      ANY      ANY    /_profiler/search_bar              web_profiler.controller.profiler::searchBarAction()
  _profiler_phpinfo         ANY      ANY      ANY    /_profiler/phpinfo                 web_profiler.controller.profiler::phpinfoAction()
  _profiler_xdebug          ANY      ANY      ANY    /_profiler/xdebug                  web_profiler.controller.profiler::xdebugAction()
  _profiler_search_results  ANY      ANY      ANY    /_profiler/{token}/search/results  web_profiler.controller.profiler::searchResultsAction()
  _profiler_open_file       ANY      ANY      ANY    /_profiler/open                    web_profiler.controller.profiler::openAction()
  _profiler                 ANY      ANY      ANY    /_profiler/{token}                 web_profiler.controller.profiler::panelAction()
  _profiler_router          ANY      ANY      ANY    /_profiler/{token}/router          web_profiler.controller.router::panelAction()
  _profiler_exception       ANY      ANY      ANY    /_profiler/{token}/exception       web_profiler.controller.exception_panel::body()
  _profiler_exception_css   ANY      ANY      ANY    /_profiler/{token}/exception.css   web_profiler.controller.exception_panel::stylesheet()
  accueil_index             ANY      ANY      ANY    /                                  App\Controller\AccueilController::indexAction()
  sandbox_overview          ANY      ANY      ANY    /sandbox/overview                  App\Controller\Sandbox\OverviewController::indexAction()
  sandbox_overview_hello2   ANY      ANY      ANY    /sandbox/overview/hello2           App\Controller\Sandbox\OverviewController::hello2Action()
  sandbox_overview_hello3   ANY      ANY      ANY    /sandbox/overview/hello3           App\Controller\Sandbox\OverviewController::hello3Action()
  sandbox_overview_hello4   ANY      ANY      ANY    /sandbox/overview/hello4           App\Controller\Sandbox\OverviewController::hello4Action()
  sandbox_prefix            ANY      ANY      ANY    /sandbox/prefix                    App\Controller\Sandbox\PrefixController::indexAction()
  sandbox_prefix_hello2     ANY      ANY      ANY    /sandbox/prefix/hello2             App\Controller\Sandbox\PrefixController::hello2Action()
  sandbox_prefix_hello3     ANY      ANY      ANY    /sandbox/prefix/hello3             App\Controller\Sandbox\PrefixController::hello3Action()
  sandbox_prefix_hello4     ANY      ANY      ANY    /sandbox/prefix/hello4             App\Controller\Sandbox\PrefixController::hello4Action()
 ------------------------- -------- -------- ------ ---------------------------------- ------------------------------------------------------------

serveur$
```

Si vous n'avez pas les 9 routes, vraisemblablement le même nom interne est utilisé deux fois.

3.3 Routes avec variables

Pour l'instant nous avons vu des routes avec des URLs fixes (ou constantes). Il est possible d'avoir des parties des URLs qui sont variables. Ainsi une action ne gérera plus une seule URL, mais un groupe d'URLs.

3.3.1 Création d'un nouveau contrôleur

Travail à effectuer

Dans un premier temps, vous allez construire un contrôleur dédié pour tester les routes (avec) variables.

Créez un nouveau contrôleur *Route* dans le répertoire *Sandbox*. Le préfixe de toutes les routes sera */sandbox/route* et le préfixe de tous les noms internes sera *sandbox_route*.

Pour créer le contrôleur, soit on fait un copier-coller d'un contrôleur existant et on le modifie, soit on utilise la console toujours avec l'option *--no-template* (solution que nous allons adopter) :

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

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

 created: src/Controller/Sandbox/RouteController.php

  Success!

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

Le répertoire pour les vues sera *templates/Sandbox/Route*.
Nous travaillerons dans ce contrôleur jusqu'à nouvel ordre.

Créez ce contrôleur sans oublier de préciser les préfixes. Pour l'instant laissez l'action *index* (que nous supprimerons rapidement) en remplaçant la *JsonResponse* par une *Response*.

Le code résultant peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/f78192458cc54d285fc1f5721b3a8dac29ce1048/src/Controller/Sandbox/RouteController.php

Voici pour information le code du contrôleur :

Sandbox/RouteController.php

```php
<?php

namespace App\Controller\Sandbox;

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

#[Route('/sandbox/route', name: 'sandbox_route')]
class RouteController extends AbstractController
{
    #[Route('', name: '')]
    public function indexAction(): Response
    {
        return new Response('<body>RouteController</body>');
    }
}
```

3.3.2 Action avec une variable simple

Il est possible d'avoir une partie de l'URL qui est laissée à la liberté de l'internaute. Pour cela :

- on précises dans l'annotation la partie variable entre accolades
- la valeur récupérée dans l'URL est injectée automatiquement en paramètre de l'action

Travail à effectuer

Il s'agit de recopier (et comprendre !) le code ci-dessous, et de le tester.

Nous allons rajouter une action (nommée *withVariable*) illustrant le cas. Pour l'instant nous construisons directement une *Response* dans la méthode.

Nous en profitons pour supprimer l'action *index* qui n'apporte aucune nouvelle information.

Le code complet du contrôleur peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/a6be158d11bd26083b82cb1986ecea21d353e568/src/Controller/Sandbox/RouteController.php

Voici le code de l'action ajoutée :

RouteController.php (extrait)

```
12      #[Route(
13          '/with-variable/{age}',
14          name: '_with_variable'
15      )]
16      public function withVariableAction($age): Response
17      {
18          return new Response('<body>Route::withVariable : age = ' . $age . '</body>');
19      }
```

Remarques

- *lignes 12 à 15* : les attributs ont été présentés sur plusieurs lignes : ce sera indispensable lorsqu'il y en aura plus.
- *lignes 13 à 14* : rappelez-vous que les préfixes sont précisés dans les attributs avant la déclaration de la classe.
- *ligne 14* : le nom interne aurait pu être *_with_variable_age*.
- *lignes 13 et 16* : la partie variable de l'URL est après le dernier "/". L'identifiant entre accolades (le nom est purement arbitraire) doit être le même que le paramètre de la méthode.
- l'internaute peut écrire presque ce qu'il désire après le dernier "/", et ce qu'il écrira se retrouvera automatiquement dans le paramètre *$age*.
  Pour l'instant il n'y a aucune contrainte sur ce qu'on peut écrire.
- *ligne 16* : notez que le paramètre n'est pas typé. Cela sera fait ultérieurement, à la fois dans les annotations et dans la signature.
- *ligne 18* : une *Response* est créée directement pour éviter l'écriture d'un Twig [1].

Testez cette route avec de multiples valeurs pour l'âge : nombre, nombre négatif, mot :

- http://localhost:8000/sandbox/route/with-variable/15
- http://localhost:8000/sandbox/route/with-variable/-15
- http://localhost:8000/sandbox/route/with-variable/colibri

Essayez à nouveau ces 3 URLs en ayant typé le paramètre *$age* en *int*. Pour la troisième URL, lisez bien le message d'erreur et dans la toolbar allez voir l'entrée "Routing" et les routes testées par Symfony. Puis enlevez le typage du paramètre.

Essayez cette route avec une URL plus longue (un slash supplémentaire) pour vérifier qu'une erreur est déclenchée :

- http://localhost:8000/sandbox/route/with-variable/oiseau/colibri

Enfin essayez cette route sans passer de valeur pour vérifier qu'une erreur est déclenchée :

- http://localhost:8000/sandbox/route/with-variable

La partie variable est donc (pour l'instant) obligatoire.

Dans les deux derniers cas, passez par la toolbar pour aller dans l'entrée "Routing" du menu et visualiser les routes testées par Symfony.

3.4 Routes avec variables et valeurs par défaut

Il est possible de ne pas remplir la (ou les) partie(s) variable(s) et de proposer une valeur par défaut.

Travail à effectuer

Il s'agit de recopier (et comprendre !) le code ci-dessous, et de le tester.

Nous allons rajouter une action (nommée *withDefault*) illustrant le cas. Nous construisons toujours directement une *Response* dans la méthode.

1. par pure flemme en l'occurrence, et aussi parce que ce qui nous intéresse est uniquement le passage de paramètres.

Le code complet du contrôleur peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/f030c75c487a7a3f793ab071070b926d45277ddc/src/Controller/Sandbox/RouteController.php

Voici le code de l'action ajoutée :

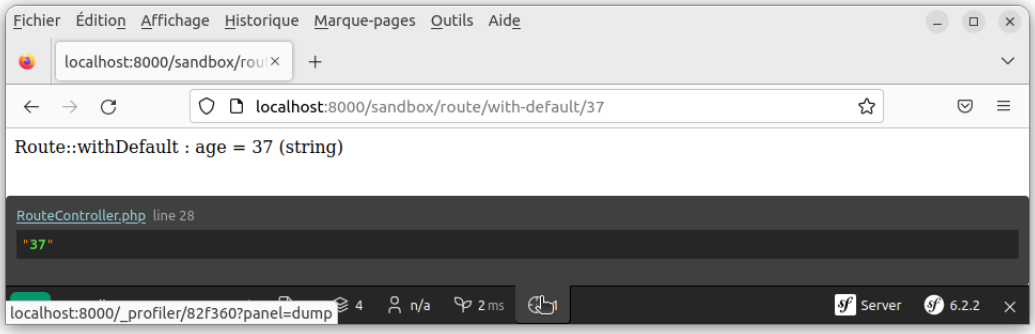
RouteController.php (extrait)

```
21      #[Route(
22          '/with-default/{age}',
23          name: '_with_default',
24          defaults: ['age' => 18],
25      )]
26      public function withDefaultAction($age): Response
27      {
28          dump($age);
29          return new Response('<body>Route::withDefault : age = ' . $age . ' (' . gettype($age) . ')</body>');
30      }
```

Remarques

- *lignes 21 à 25* : écrire les attributs sur plusieurs lignes devient bien plus lisible que sur une seule.
- *ligne 24* : comme il peut y avoir plusieurs parties variables dans une URL (cf. plus loin dans le document), l'attribut *defaults* est un tableau.
- *ligne 24* : on note la virgule en fin de ligne (comme dans les tableaux PHP). L'avantage est que si on ajoute (ou enlève) une ligne on ne se préoccupe plus de gérer la virgule sur la dernière ligne.
- *ligne 28* : pour la fonction *dump*, cf. ci-dessous.
- *ligne 29* : il est est de plus en plus déraisonnable de mettre le code HTML directement dans le corps de la méthode.

On en profite pour introduire la fonction *dump* (qui est aussi disponible dans les templates twig). Elle permet d'afficher des informations sans perturber l'affichage d'une page : on aperçoit l'apparition d'une petite cible dans la toolbar et il suffit de passer la souris dessus.



Note : *dump* sert uniquement à débugguer [2] et non à faire un affichage définitif.

Reprenez les cas de la section précédente :

- http://localhost:8000/sandbox/route/with-default/15
- http://localhost:8000/sandbox/route/with-default/-15
- http://localhost:8000/sandbox/route/with-default/colibri
- http://localhost:8000/sandbox/route/with-default/oiseau/colibri
- http://localhost:8000/sandbox/route/with-default

et vérifiez ainsi qu'il est possible désormais de ne plus préciser le paramètre.

Remarquez que, pour l'instant, *$age* est une chaîne de caractères et non un entier (sauf pour la valeur par défaut). Et on ne peut pas encore typer le paramètre de l'action car une chaîne quelconque pourrait être mise dans l'URL.

Essayez l'URL suivante :

- http://localhost:8000/sandbox/route/with-variable/

On remarque que l'URL avec le *training slash* est d'abord réécrite sans le "/".

2. comme *print_r* ou *var_dump*

3.5 Routes avec contraintes

Dans la route précédente, une liberté totale est laissée à l'internaute quant à la partie variable de l'URL.

Or clairement dans notre cas nous attendons un nombre positif. Cela laisse présager un tests en début de chaque action pour vérifier que la partie variable est bien formée.
Et lorsqu'il y aura plusieurs parties variables avec des contraintes plus précises (un numéro de mois correct, une extension de fichier image correcte, ...) les tests prendront rapidement une place importante dans le code.

Il est possible de mettre, dans l'annotation, une contrainte via une expression régulière [3].

Si l'URL ne respecte pas la contrainte, l'action n'est pas appelée et une erreur 404 est levée.
Il faut insister : on n'entre même pas dans la méthode, et donc il est inutile de faire des tests (du style *is_numeric*) en début de fonction.

Travail à effectuer

Il s'agit de recopier (et comprendre !) le code ci-dessous, et de le tester.

Nous allons rajouter une action (nommée *withConstraint*) illustrant le cas. Nous construisons toujours directement une *Response* dans la méthode.

Le code complet du contrôleur peut être visualisé à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/4ae63a80447b22d610bc3acb7b7f37bb0088b142/src/Controller/Sandbox/RouteController.php

Voici le code de l'action ajoutée :

RouteController.php (extrait)

```
32      #[Route(
33          '/with-constraint/{age}',
34          name: '_with_constraint',
35          requirements: ['age' => '[0-9]+'],
36          defaults: ['age' => 18],
37      )]
38      public function withConstraintAction($age): Response
39      {
40          dump($age);
41          return new Response('<body>Route::withConstraint : age = ' . $age . ' (' . gettype($age) . ')</body>');
42      }
```

Remarques

- *lignes 32 à 37* : l'ordre des paramètres a une importance, même si cela semble fonctionner en cas de non-respect.
- *ligne 35* : de même que pour *defaults*, *requirements* est un tableau.
- *ligne 35* : `[0-9]` peut être remplacé par `\d`
- *ligne 35* : l'expression régulière pourrait avantageusement être remplacée par : `0|[1-9]\d*`
  En effet avec la première version, une expression du style "00032" est acceptée.
  Plus on bloque de cas avec l'expression régulière, moins il y a de tests à faire dans le corps de la méthode.

Testez les URLs suivantes :

- http://localhost:8000/sandbox/route/with-constraint/0
- http://localhost:8000/sandbox/route/with-constraint/15
- http://localhost:8000/sandbox/route/with-constraint/0015
- http://localhost:8000/sandbox/route/with-constraint
- http://localhost:8000/sandbox/route/with-constraint/-15
- http://localhost:8000/sandbox/route/with-constraint/colibri

La troisième est (pour l'instant) correcte.
Les deux dernières génèrent désormais une erreur 404.

Pour une des deux dernières URLs, cliquez dans la toolbar, allez dans le menu "Routing" et regardez le message d'erreur sur la route *sandbox_route_with_constraint* qui est plus explicite :

3. Les expressions régulières ou *regex* sont incontournables en informatique et doivent être connues.

Travail à effectuer

Il s'agit de parfaire les restrictions :
- mettre l'expression régulière plus restrictive [a] : `0|[1-9]\d*`
- typer le paramètre de l'action en *int*

a. un tutoriel vidéo : https://www.youtube.com/watch?v=dinW2QTSNl4

Le nouveau code du contrôleur est récupérable à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/1c89a450eae075ae3e04d369245e0ac28e789a3d/src/Controller/Sandbox/RouteController.php

Voici les changements effectués :

RouteController.php (extrait)

```
32      #[Route(
33          '/with-constraint/{age}',
34          name: '_with_constraint',
35          requirements: ['age' => '0|[1-9]\d*'],
36          defaults: ['age' => 18],
37      )]
38      public function withConstraintAction(int $age): Response
39      {
```

Le paramètre *$age* est maintenant systématiquement de type entier.

Testez les URLs suivantes qui fonctionnent :
- http://localhost:8000/sandbox/route/with-constraint/0
- http://localhost:8000/sandbox/route/with-constraint/15

Et les URLs suivantes qui ne fonctionnent plus :
- http://localhost:8000/sandbox/route/with-constraint/00
- http://localhost:8000/sandbox/route/with-constraint/015

3.6 exercice *test1*

Travail à effectuer

Avec les explications vous devez écrire :
- une action, avec ses attributs, contenant des parties variables
- la vue Twig qui affiche les paramètres de l'action

Créez une route qui prend 4 paramètres :
- *year*
- *month*
- *filename*
- *ext*

Une URL ressemblera à : http://localhost:8000/sandbox/route/test1/2014/11/image.jpg

Pour l'instant aucune vérification n'est faite sur les paramètres.

Créez une action et une vue qui affiche ces quatre paramètres. La vue pourra être réutilisée telle quelle pour les 3 exercices suivants (actions *test2*, *test3* et *test4*).

Faites différents tests d'URL, avec certaines erronées, et passez par la toolbar pour vérifier les routes testées par Symfony.

Par exemple testez les URLs suivantes :
- http://localhost:8000/sandbox/route/test1/2014/11/image.jpg
- http://localhost:8000/sandbox/route/test1/2014/11/image.txt

- http://localhost:8000/sandbox/route/test1/2014/15/image.jpg
- http://localhost:8000/sandbox/route/test1/anmil/nov/8.55
- http://localhost:8000/sandbox/route/test1/2014/15/image.jpg/ (notez la suppression automatique du *trailing slash* qui est une redirection cachée)

qui devraient marcher.

Et testez les URLs suivantes :

- http://localhost:8000/sandbox/route/test1/2014/11/image.
- http://localhost:8000/sandbox/route/test1/2014/11/image
- http://localhost:8000/sandbox/route/test1/2014/15/image.jpg/autre

qui devraient déclencher une erreur.

Vous pouvez comparer votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8c5a54d7acb3405f3db5bd9bae7ebd87ff5c33a8/src/Controller/Sandbox/RouteController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8c5a54d7acb3405f3db5bd9bae7ebd87ff5c33a8/templates/Sandbox/Route/test1234.html.twig

Voici des extraits :

src/Controller/Sandbox/RouteController.php (extrait)

```
44      #[Route(
45          '/test1/{year}/{month}/{filename}.{ext}',
46          name: '_test1',
47      )]
48      public function test1Action($year, $month, $filename, $ext): Response
49      {
50          $args = array(
51              'title' => 'test1',
52              'year' => $year,
53              'month' => $month,
54              'filename' => $filename,
55              'ext' => $ext,
56          );
57          return $this->render('Sandbox/Route/test1234.html.twig', $args);
58      }
```

templates/Sandbox/Route/test1234.html.twig (extrait)

```
14      <body>
15          <h1>Route::{{ title }}</h1>
16          <table>
17              <tr><th>year</th>  <td>{{ year }}</td></tr>
18              <tr><th>month</th> <td>{{ month }}</td></tr>
19              <tr><th>filename</th><td>{{ filename }}</td></tr>
20              <tr><th>ext</th>   <td>{{ ext }}</td></tr>
21          </table>
22          {{ dump(year, month, filename, ext) }} {# sont-ce des chaînes ? #}
23      </body>
```

3.7 exercice *test2*

Travail à effectuer

Avec les explications vous devez écrire une route avec des restrictions sur les arguments.

Créez une route qui reprend les mêmes quatre paramètres que *test1* mais qui s'appelle *test2*.

Une route ressemblera à : /sandbox/route/test2/2014/11/image.jpg

Cette fois-ci, les paramètres sont vérifiés :

- *year* : sur 4 chiffres maximum, strictement positive et ne doit pas commencer par 0.
- *month* : doit être composé de chiffres, être correct et peut commencer par 0.
- *filename* : ne contient que des lettres et le signe "-"
- *ext* : exclusivement jpg, jpeg, png et ppm

Aucune valeur par défaut n'est proposée.

On peut brancher cette action sur la même vue que *test1*.

De nouveaux essayez plusieurs URL, et en cas de non respect des restrictions regardez les messages de la toolbar qui peuvent être assez explicites.

Voici un ensemble d'URLs qui fonctionnent :

- http://localhost:8000/sandbox/route/test2/2014/11/image.jpg
- http://localhost:8000/sandbox/route/test2/2014/12/image.jpeg
- http://localhost:8000/sandbox/route/test2/2014/3/image-test.ppm
- http://localhost:8000/sandbox/route/test2/2014/03/——.png

Et un autre ensemble d'URLs qui échouent (et qui marchaient avec *test1*) :

- http://localhost:8000/sandbox/route/test2/2014/11/image.txt (pb. extension)
- http://localhost:8000/sandbox/route/test2/2014/15/image.jpg (pb. mois incorrect)
- http://localhost:8000/sandbox/route/test2/2014/nov/image.jpg (pb. mois incorrect)
- http://localhost:8000/sandbox/route/test2/2014/11/image3.jpg (pb. nom avec un chiffre)
- http://localhost:8000/sandbox/route/test2/0523/11/image.jpg (pb année commence par 0)
- http://localhost:8000/sandbox/route/test2/21578/11/image.jpg (pb. année trop grande)

Vous pouvez comparer votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/b633e32b2de997875ab4831a1ad21b534b7ea024/src/Controller/Sandbox/RouteController.php

Voici un extrait :

Sandbox/RouteController.php (extrait)

```
60      #[Route(
61          '/test2/{year}/{month}/{filename}.{ext}',
62          name: '_test2',
63          requirements: [
64              'year' => '[1-9]\d{0,3}',
65              'month' => '(0?[1-9])|(1[0-2])',
66              'filename' => '[-a-zA-Z]+',
67              'ext' => 'jpg|jpeg|png|ppm',
68          ],
69      )]
70      public function test2Action(int $year, int $month, string $filename, string $ext): Response
71      {
72          $args = array(
73              'title' => 'test2',
```

On remarque qu'on en a profité pour typer les paramètres.

3.8 exercice *test3*

Travail à effectuer

Avec les explications vous devez écrire une route avec des des valeurs par défaut sur des arguments.

Créez une troisième route, nommée *test3* identique à la précédente.

Mais cette fois-ci il y a une valeur par défaut pour *ext*, il s'agit de *png*.

On peut brancher cette action sur la même vue que *test2*.

De nouveaux essayez plusieurs URL, dont notamment :

- */sandbox/route/test3/2014/11/image.* (avec le point)
- */sandbox/route/test3/2014/11/image*
- */sandbox/route/test3/2014/15/image/* (avec le slash)

La première URL ne fonctionne pas alors que la seconde oui : le "." se comporte comme un "/".
Pour la troisième URL nous avons déjà vu le comportement : elle est réécrite sans le training slash et donc on est ramené à la seconde URL.

Question subsidiaire : comme valeur par défaut, peut-on proposer une extension non valide, comme *gif* par exemple ?

Vous pouvez comparer votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/d68d416a291a47a0fb8f4f90a31546c24701e5e3/src/Controller/Sandbox/RouteController.php

Voici un extrait :

Sandbox/RouteController.php (extrait)

```php
    #[Route(
        '/test3/{year}/{month}/{filename}.{ext}',
        name: '_test3',
        requirements: [
            'year' => '[1-9]\d{0,3}',
            'month' => '(0?[1-9])|(1[0-2])',
            'filename' => '[-a-zA-Z]+',
            'ext' => 'jpg|jpeg|png|ppm',
        ],
        defaults: [
            'ext' => 'png',  // on peut proposer une valeur non valide comme "gif"
        ],
    )]
    public function test3Action(int $year, int $month, string $filename, string $ext): Response
    {
        $args = array(
            'title' => 'test3',
```

3.9 exercice *test4*

Travail à effectuer

Avec les explications vous devez écrire une quatrième route pour une étude plus approfondie des des valeurs par défaut sur des arguments.

Enfin créez une quatrième route, nommée *test4* identique à la précédente.

Mais il y a également une valeur par défaut pour *month*, il s'agit de *1*.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/232eb0dc940d8f729960a4d777453f88dbeed26d/src/Controller/Sandbox/RouteController.php

Arrivez-vous à utiliser ce paramètre par défaut ? [4]
Les paramètres ayant une valeur par défaut sont obligatoirement ceux en fin, sinon la résolution des URL serait ambiguë.

Ajoutez également une valeur par défaut pour *filename*, il s'agit de *picture*.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/2d4d1c6b6e9a79b26bb08df370a960bc5fc8a2e3/src/Controller/Sandbox/RouteController.php

Et dans ce cas, tout fonctionne comme attendu.

3.10 exercice *test4* encore

Travail à effectuer

Avec les explications vous devez écrire une cinquième route ... qui ne sert à rien.

Enfin (bis repetita) créez une cinquième route route, dont l'URL support est :
/sandbox/route/test4/{year}
c'est bien *test4* comme la précédente et non pas *test5*.
On reprend donc la route précédente sauf que les paramètres *month*, *filename* et *ext* n'existent plus.

Le nom interne doit être différent du précédent (c'est une obligation), par exemple *test4bis*.
Et il faut une vue dédiée à cette action.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/bcc4bb194a0209c9abbd3da363398c7fc0372ab8/src/Controller/Sandbox/RouteController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/bcc4bb194a0209c9abbd3da363398c7fc0372ab8/templates/Sandbox/Route/test4bis.html.twig

Ce qui donne comme code de la nouvelle action :

Sandbox/RouteController.php (extrait)

4. Personnellement je n'y suis pas arrivé, et je doute que ce soit possible.

```php
    #[Route(
        '/test4/{year}',
        name: '_test4bis',
        requirements: [
            'year' => '[1-9]\d{0,3}',
        ],
    )]
    public function test4bisAction(int $year): Response
    {
        $args = array(
            'title' => 'test4bis',
            'year' => $year,
        );
        return $this->render('Sandbox/Route/test4bis.html.twig', $args);
    }
```

Le but est d'avoir un "conflit" entre cette route et la précédente.

Et faites des tests. Vous remarquez qu'il est impossible de déclencher cette action car elle est incluse dans (ou masquée par) la précédente [5].

En revanche, si vous permutez l'ordre des deux actions *test4* et *test4bis* dans le code du contrôleur, vous changez les priorités.
Testez la permutation des deux codes, puis revenez à l'ordre initial.

Ce n'est qu'un exercice de style, il faut absolument éviter que des routes entrent en conflit, même partiellement.

4 Paramètres injectés

Il existe, pour les actions, des paramètres "hors route", on parle d'injection de paramètres, et plus exactement d'injection de dépendances.
Ces paramètres sont donc instanciés automatiquement par Symfony.

C'est l'occasion de dire qu'une action ne doit jamais être appelée explicitement par le programmeur : outre la gestion de ces paramètres automatiques, on pourrait perdre tout un environnement d'exécution.

Cette section s'intéresse (survole plutôt) la gestion des paramètres GET et POST, des cookies et des variables de sessions, autrement dit de la gestion de certaines variables super-globales par Symfony.

4.1 Nouveau contrôleur

Travail à effectuer

Le but est de créer un nouveau contrôleur pour étudier l'injection de paramètres dans les méthodes.

Nous travaillons, toujours dans le répertoire *Sandbox* avec un nouveau contrôleur *Injection* avec ses vues dédiées (si nécessaire).

Commencez par créer ce contrôleur dont le préfixe est */sandbox/injection*. Pour l'instant il n'a pas de méthode.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/79ee65fc2645e876d79af299f2ff8d61991992b4/src/Controller/Sandbox/InjectionController.php

Soit le code (pas extrêmement passionnant) :

Sandbox/InjectionController.php

```php
<?php

namespace App\Controller\Sandbox;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

5. Ou plus exactement il est impossible de déclencher cette action en tapant une URL dans le navigateur ; mais il est possible de la déclencher par programme avec une redirection (cf. plus loin).

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

#[Route('/sandbox/injection', name: 'sandbox_injection')]
class InjectionController extends AbstractController
{
}
```

4.2 Objet Request

Travail à effectuer

Le but est de créer une action injectant l'objet *Request* et de tester quelques méthodes pour le manipuler :

- les paramètres GET de l'URL
- les paramètres POST (souvent des formulaires [a])
- les cookies

a. Mais Symfony a un module dédié aux formulaires

Ajoutez une action dans le contrôleur *Injection* nommée *unAction* dont la route correspondante est : */sandbox/injection/un*.
Dans l'immédiat il n'y pas pas de paramètre.

Si l'internaute saisit l'URL suivante :
http://localhost:8000/sandbox/injection/un?foo=999&bar=aaaa
Comment récupère-t-on les paramètres "foo" et "bar" ?

Pour cela il faut utiliser l'objet de type *Request* qui permet de manipuler les paramètres GET, les cookies, les paramètres POST (formulaire par exemple), et d'autres encore.

La meilleure explication est de visualiser le code :

Sandbox/InjectionController.php (extraits)

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

```php
#[Route('/sandbox/injection', name: 'sandbox_injection')]
class InjectionController extends AbstractController
{
    #[Route('/un', name: '_un')]
    public function unAction(Request $request): Response
    {
        dump($request);
        return new Response('<body>Injection::un</body>');
    }
}
```

Note : on choisit le type "`Symfony\Component\HttpFoundation\Request`"

Vous pouvez récupérer le code complet :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8d2d24814e01cca8e36fc18d2a68487d299e24b2/src/Controller/Sandbox/InjectionController.php

Affichez la page dans votre navigateur et jetez un rapide coup d'oeil sur le contenu de *$request* (dans la toolbar, l'icône en forme de cible).
Il est particulièrement complexe et n'est pas fait pour être utilisé directement ; il faut passer par les méthodes de l'objet, ou les méthodes de certains objets contenus dans *$request*.

Voici quelques objets contenus dans *Request* avec leurs correspondances avec les variables super-globales :

```
$_GET     →  $request->query
$_POST    →  $request->request
$_COOKIE  →  $request->cookies
...
```

Pour récupérer tous les arguments : méthode *all*, par exemple :

```php
$request->cookies->all() // i.e. $_COOKIE
```

Pour récupérer un argument : méthode *get*, par exemple :

```
$request->query->get('foo') // i.e. $_GET['foo']
```

Pour récupérer la méthode de requête HTTP :

```
$request->getMethod()
```

Savoir si la requête est en POST :

```
if ($request->isMethod('POST')) ...
```

Pour savoir si la requête est une requête Ajax :

```
if ($request->isXmlHttpRequest()) ...
```

Le but est juste d'avoir un aperçu de cet objet ; référez-vous à la documentation officielle pour une liste complète.

Ajoutez dans l'action, via des appels à *dump*, l'affichage :
- de la méthode HTTP
- de la variable GET : *foo*
- de toutes les variables GET
- de tous les cookies

Vous pouvez comparer votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/e3899b8c15bc1ed5e833cdc7fcda82c7c78e2d8c/src/Controller/Sandbox/InjectionController.php

Voici un extrait :

Sandbox/InjectionController.php (extraits)

```
13      #[Route('/un', name: '_un')]
14      public function unAction(Request $request): Response
15      {
16          dump($request->getMethod());
17          dump($request->query->get('foo'));
18          dump($request->query->all());
19          dump($request->cookies->all());
20          return new Response('<body>Injection::un</body>');
21      }
```

Testez avec plusieurs URLs, par exemple :
- http://localhost:8000/sandbox/injection/un?foo=999&bar=aaaa
- http://localhost:8000/sandbox/injection/un?bar=aaaa
- http://localhost:8000/sandbox/injection/un?nom=Shepard&prenom=Jane&monde=Mass%20Effect

En cliquant sur la toolbar, et en prenant la première entrée du menu de gauche (*"Request/Response"*), on a également accès au contenu de l'objet *Request*.

Remarque : il est préférable de remplacer les paramètres GET par des vrais paramètres Symfony, notamment car on peut effectuer des vérifications dans l'annotation de l'action correspondante.

4.3 Objet Session

Travail à effectuer

Le but est de créer une action injectant l'objet *Session* et de tester quelques méthodes pour le manipuler : ajout, modification, suppression de variables de session.

Il est également possible de récupérer un objet permettant de manipuler les sessions (i.e. `$_SESSION`) :
- l'objet à injecter est de type *Session*
- la méthode *get* récupère la valeur d'une variable
- la méthode *all* récupère toutes les valeurs
- la méthode *set* crée ou modifie la valeur d'une variable
- la méthode *remove* supprime une variable de session

Ajoutez une action dans le contrôleur *Injection* nommée *deuxAction* dont la route correspondante est : */sandbox/injection/deux* complétée d'un potentiel argument GET.

Voici le principe de l'action :

- si le paramètre GET nommé *compteur* existe, alors on affecte la variable de session *compteur* avec la valeur du paramètre.
  exemple : http://localhost:8000/sandbox/injection/deux?compteur=37
- si le paramètre GET nommé *inc* existe, alors on augmente la variable de session *compteur* d'une unité.
  c'est à dire : http://localhost:8000/sandbox/injection/deux?inc
- si le paramètre GET nommé *supp* existe, alors on détruit la variable de session *compteur*.
  c'est à dire : http://localhost:8000/sandbox/injection/deux?supp
- dans tous les cas on affiche, avec *dump*, toutes les variables de session.
- dans tous les cas on affiche, avec *dump*, la variable *$_SESSION*. Attention avec Symfony on ne manipule jamais directement *$_SESSION*, mais on passe systématiquement l'objet *Session*.

Essayez de faire le code de l'action. N'hésitez à regarder le code ci-dessous si vous être bloqué.

Vous pouvez comparer votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/4cf7518ce12240f721e6daea55f213145127a11c/src/Controller/Sandbox/InjectionController.php

Voici directement le code :

Sandbox/InjectionController.php (extraits)

```
8  use Symfony\Component\HttpFoundation\Session\Session;
```

```
24     #[Route('/deux', name: '_deux')]
25     public function deuxAction(Request $request, Session $session): Response
26     {
27         // note : on peut aussi récupérer la session avec : $request->getSession();
28         if ($request->query->get('compteur') !== null)
29             $session->set('compteur', $request->query->get('compteur'));
30         elseif ($request->query->get('inc') !== null)
31             $session->set('compteur', $session->get('compteur') + 1);
32         elseif ($request->query->get('supp') !== null)
33             $session->remove('compteur');
34         dump($session->all());
35         dump($_SESSION);
36         return new Response('<body>Injection::un</body>');
37     }
```

Note : on choisit le type "`Symfony\Component\HttpFoundation\Session\Session`"
Note : on peut aussi récupérer la session via l'objet *Request* :

```
$request->getSession()
```

Il ne faut pas/plus manipuler directement la variable *$_SESSION* : Symfony la réorganise à sa manière ; il faut utiliser systématiquement l'objet *Session*.

5 Déclencher une erreur 404

Travail à effectuer

Le but est de créer une action déclenchant explicitement une erreur 404.

On revient dans le contrôleur *Route*.

Toutes les contraintes d'un paramètre ne peuvent pas être décrites dans les annotations. Le code d'une action peut lui même avoir besoin de déclencher une erreur 404 avec le code suivant :

```
throw new NotFoundHttpException('...');
```

Note : le type est "`Symfony\Component\HttpKernel\Exception\NotFoundHttpException`".

Ajoutez une action dans le contrôleur *Route* avec l'URL : */sandbox/route/permis/{age}*
La variable *age* doit être composée de chiffres exclusivement et c'est la seule contrainte que l'on met dans les annotations.

Si l'âge est inférieur à 18 ans, levez explicitement une exception.

Les deux URLs suivantes génèrent une erreur 404 :

- http://localhost:8000/sandbox/route/permis/toto
- http://localhost:8000/sandbox/route/permis/15

La première est déclenchée automatiquement par Symfony du fait des annotations, la seconde explicitement par le développeur.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/fb3cc8905590c48ceaac8199c5f86f0228b072c8/src/Controller/Sandbox/RouteController.php

C'est à dire le code de la méthode :

Sandbox/RouteController.php (extrait)

```
151     #[Route(
152         '/permis/{age}',
153         name: '_permis',
154         requirements: [
155             'age' => '\d+',
156         ],
157     )]
158     public function permisAction(int $age): Response
159     {
160         if ($age < 18)
161             throw new NotFoundHttpException('Vous n\'êtes pas assez âgé !');
162         return new Response('<body>Route::permis : age = ' . $age . ' (&ge; 18)</body>');
163     }
```

6 Les redirections

On reste dans le contrôleur *Sandbox/Route*.

Après le traitement d'une action, il peut être nécessaire de faire une redirection (équivalent de la fonction PHP : `header('Location: ...')`).

6.1 Redirection simple

Travail à effectuer

Le but est de créer une action faisant une redirection vers une route ne prenant pas de paramètre.

Voici un exemple nécessitant une redirection : après avoir supprimé un livre dans la base de données, il n'y a aucune vue à afficher [6]. Le plus habituel est de faire une redirection vers l'action affichant l'ensemble des livres.

Une redirection est une réponse HTTP comme une autre, et donc un objet de type *Response* à retourner par une action. Et les contrôleurs fournissent une méthode dédiée : *redirectToRoute*.

Voici la syntaxe dans une action :

```
return $this->redirectToRoute('nom interne de la route');
```

On note bien le *"return"* : c'est un objet *Response* retourné par l'action.

Lorsqu'on fait une redirection, on ne précise jamais une URL mais toujours le nom interne d'une route [7]. C'est pour cela que les noms internes doivent être uniques (et explicites).

Par exemple pour faire une redirection vers la deuxième route du contrôleur *Prefix* :

```
return $this->redirectToRoute('sandbox_prefix_hello2');
```

6. Une vue affichant "le livre a été supprimé" n'est pas très intéressante/ergonomique et les messages flash, que nous étudierons plus loin, remplissent bien mieux ce rôle.
7. sauf s'il s'agit d'une page extérieure au site

Écrivez une action avec l'URL :
http://localhost:8000/sandbox/route/redirect1
qui fait une redirection sur la route "sandbox_prefix_hello4" (ou une autre route qui ne prend pas de paramètre).

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/c35291401789abbbc2a69c56b54bc5b62a62525b/src/Controller/Sandbox/RouteController.php

C'est à dire le code de la méthode :

Sandbox/RouteController.php (extrait)

```
165     #[Route('/redirect1', name: '_redirect1')]
166     public function redirect1Action(): Response
167     {
168         return $this->redirectToRoute('sandbox_prefix_hello4');
169     }
```

Note importante : à la place de faire une redirection, on pourrait être tenté de lancer directement la vue de l'action que l'on vise par la redirection, i.e. plutôt que le *redirectToRoute* faire une instruction du style :

```
return $this->render('Sandbox/Prefix/hello4.html.twig'); // NON
```

En fait ce n'est pas correct (voire pas possible) :

- la vue peut attendre des paramètres
- et en l'occurrence c'est le cas puisque la vue *hello4.html.twig* attend un titre et un tableau de jeux.
- l'action courante (*redirect1*) soit ne connaît pas la valeur des paramètres et on est coincé, soit serait obligé de dupliquer une partie du code de la méthode `PrefixController::hello4Action` ce qui est prohibé en programmation.

Note importante (bis) : à la place de faire une redirection, on pourrait être tenté d'appeler directement la méthode cible de la redirection, i.e. plutôt que le *redirectToRoute* faire une instruction du style :

```
return \App\Controller\Sandbox\PrefixController::hello4Action(...); // NON
```

En fait ce n'est pas correct (voire pas possible) :

- (il faudrait vérifier que c'est techniquement possible, je n'ai pas essayé)
- la méthode peut prendre des paramètres qu'on ne maîtrise pas : les objets injectés (*Request*, *Session*, ...) que l'on serait en peine de transmettre

Bref il faut absolument exécuter le code de l'action cible de la redirection en appelant la méthode *redirectToRoute*.

6.2 Redirection avec paramètres

Travail à effectuer

Le but est de créer une action faisant une redirection vers une route prenant des paramètres.

Il est possible d'effectuer une redirection vers une route qui prend des paramètres. La syntaxe est alors :

```
return $this->redirectToRoute('nom interne', <tableau de paramètres>);
```

Par exemple pour effectuer une redirection vers la première route du contrôleur *Route* :

```
return $this->redirectToRoute('sandbox_route_with_variable', array('age' => 18));
```

Le tableau contient autant de cases qu'il y a de paramètres à passer à la route ; et les clés des cases sont les noms des paramètres de la route visée.

Écrivez une action avec l'URL :
http://localhost:8000/sandbox/route/redirect2
qui fait une redirection vers la route de la section "3.8. exercice *test3* ".
Choisissez des valeurs en dur pour les 4 paramètres. Essayez de ne pas renseigner le quatrième paramètre pour vérifier si la valeur par défaut est bien utilisée.

Vous pouvez vérifier votre code avec :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/0f7773f07b035145c1a8dd409b95ec4862250dab/src/Controller/Sandbox/RouteController.php

C'est à dire le code de la méthode :

Sandbox/RouteController.php (extrait)

```
171     #[Route('/redirect2', name: '_redirect2')]
172     public function redirect2Action(): Response
173     {
174         $params = array(
175             'year' => 1815,
176             'month' => 12,
177             'filename' => 'ada',
178             'ext' => 'ppm',              // tester en commentant la ligne, puis en mettant une extension interdite
179         );
180         return $this->redirectToRoute('sandbox_route_test3', $params);
181     }
```

6.3 Interception des redirections

Travail à effectuer

Le but est de créer une action pour illustrer l'interception des redirections.

Écrivez une action avec l'URL :
http://localhost:8000/sandbox/route/redirect3
qui fait la même redirection que "redirect1". En plus, avant la redirection, faites un *dump* d'une donnée quelconque.

Vous pouvez utiliser le code récupérable à :

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

C'est à dire le code de la méthode :

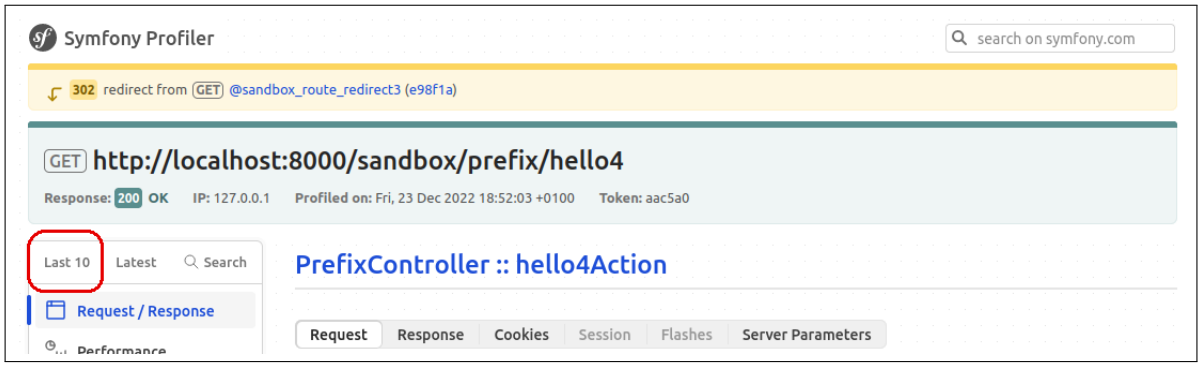
Sandbox/RouteController.php (extrait)

```
183     #[Route('/redirect3', name: '_redirect3')]
184     public function redirect3Action(): Response
185     {
186         dump('bonjour');
187         return $this->redirectToRoute('sandbox_prefix_hello4');
188     }
```

Après la redirection, le *dump* n'apparaît pas dans la toolbar. En effet le *dump* ne dure que le temps d'exécution d'une route ; une redirection relance Symfony complètement et donc une nouvelle route.

Or nous aimerions bien étudier l'état de l'action avant redirection : le *dump* n'est qu'un prétexte, mais on pourrait imaginer qu'il y ait des requêtes SQL ou des variables de session à visualiser.

Une première solution serait d'utiliser la toolbar est de naviguer dans l'historique de la navigation (la toolbar stocke les 10 derniers liens utilisés), mais c'est très fastidieux.
Si vous êtes intéressé, voici la zone de l'interface concernée :



Symfony permet de faire une pause avant la redirection. Pour cela il faut éditer le fichier :
config/packages/web_profiler.yaml
et mettre le paramètre *intercept_redirects* à *true* :

config/packages/web_profiler.yaml (extrait)

```yaml
when@dev:
    web_profiler:
        toolbar: true
        intercept_redirects: true
```

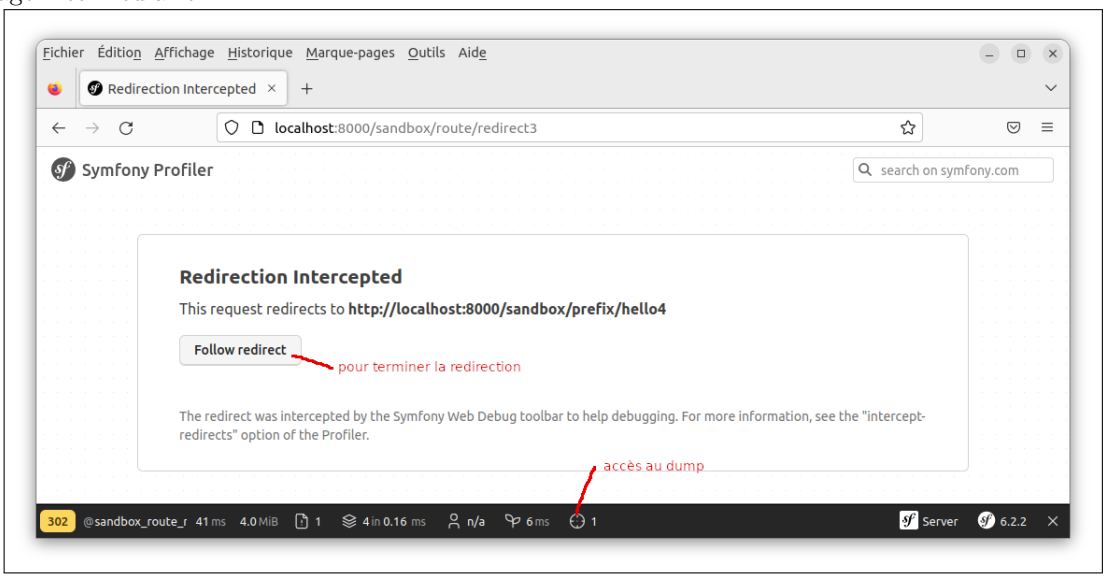
Vous pouvez récupérer les deux fichiers :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/91ab4ca800132e68136727f9fd114ad7f4d2a191/src/Controller/Sandbox/RouteController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/91ab4ca800132e68136727f9fd114ad7f4d2a191/config/packages/web_profiler.yaml

Regardez désormais le comportement de votre route :

- vous avez le temps de regarder le contexte de l'action courante (via la toolbar par exemple) sur une page intermédiaire :



- avant de déclencher la redirection manuellement et d'arriver sur la page cible.

7 Messages flash

Travail à effectuer

Lire l'introduction avant la mise en pratique.

On revient dans le contrôleur *Injection*.

Ce qui va être étudié dans cette section, c'est le fait de positionner des messages dans une action pour un affichage ultérieur, c'est à dire dans une autre action.
Ce type de messages s'appellent des messages flash.

Voici un scénario classique :

- on appelle l'action *delete* de suppression d'un livre dans la base de données
- l'action *delete* fait la suppression
- l'action *delete* positionne un message flash
- l'action *delete* fait une redirection vers la route *list* qui affiche tous les livres
- l'action *list* lance sa vue twig
- la vue affiche les messages flash [8]

Symfony propose un outil pour gérer automatiquement ce mécanisme [9] qui utilise les variables de session [10].

Les sous-sections suivantes expliquent le fonctionnement de ces messages.

8. En réalité nous verrons dans un prochain TP que toutes les vues affichent les messages flash.
9. Ce ne serait pas très compliqué de les programmer soi-même, mais ce serait long et fastidieux.
10. Il n'y pas vraiment le choix si l'on veut rester efficace.

7.1 Création des messages flash

Travail à effectuer

Le but est d'écrire une action créant des messages flash.
Il suffit de suivre [a] les indications au fur et à mesure pour avoir un code effectif.

a. et comprendre !

Les messages flash sont stockés en session, et il y a deux manière de les créer :

- on injecte dans en paramètre de l'action un objet de type *Session* et :
  - on appelle la méthode *getFlashBag* qui renvoie l'objet gérant les messages flash
  - et sur cet objet on appelle la méthode *add*
- les contrôleurs possèdent la méthode *addFlash* qui fait les deux opérations ci-dessus de manière transparente.

Les messages flash sont regroupés en catégories. Une catégorie est une simple étiquette (une chaîne) laissée au choix du programmeur. Aussi lorsqu'on crée un message (méthode *add* ou *addFlash*) il faut préciser le message et la catégorie.

Créons une action qui illustre les deux manières d'ajouter un message.

Vous pouvez récupérer le code complet du contrôleur à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/461167c204fd3329b8968a872dbe33f87b1fba09/src/Controller/Sandbox/InjectionController.php

Et voici directement le code de l'action ajoutée :

Sandbox/InjectionController.php (extrait)

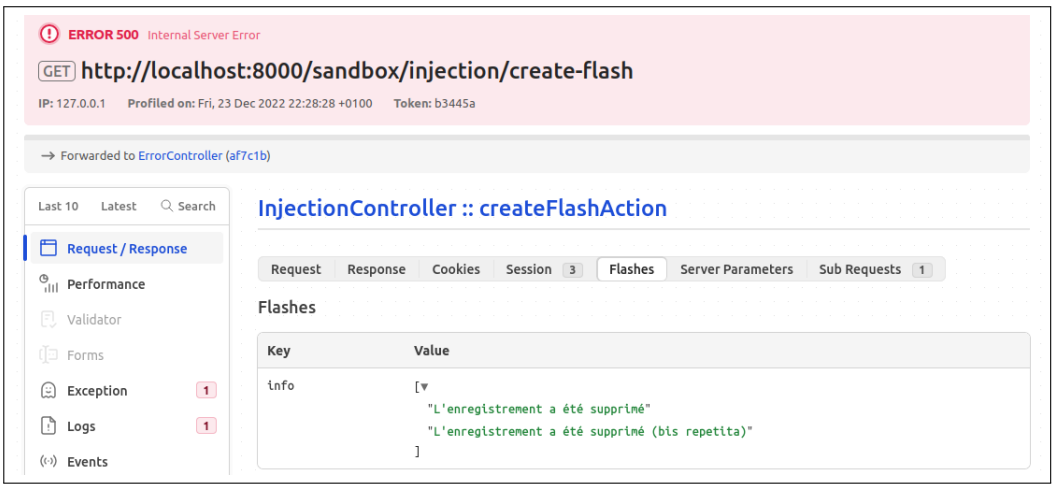
```
39      #[Route('/create-flash', name: '_create_flash')]
40      public function createFlashAction(Session $session): Response
41      {
42          // par exemple cette action supprime une entrée dans la base de données
43          $session->getFlashBag()->add('info', 'L\'enregistrement a été supprimé');
44          $this->addFlash('info', 'L\'enregistrement a été supprimé (bis repetita)');
45          return $this->redirectToRoute('sandbox_injection_display_flash');
46      }
```

Explications :

- *lignes 43* vs. *44* : les deux lignes sont strictement équivalentes. La seconde est plus simple, et notamment elle ne nécessite pas d'injecter la session dans les paramètres de l'action.
- On insiste sur le fait que *info* est une étiquette libre. On peut créer autant de catégories que l'on souhaite.
- *ligne 45* : on fait une redirection vers une autre route qui n'existe pas pour l'instant.

Ceci dit on peut tout de même tester la route :

- on tombe bien entendu sur une erreur de Symfony.
- mais on peut visualiser les deux messages :
  - en cliquant sur la toolbar
  - en choisissant l'entrée *Request/Response*
  - et l'onglet *Flashes*



Relancez l'URL deux fois et visualisez les messages flash : on s'aperçoit qu'ils sont en triple. En effet tant qu'ils n'ont pas été lus (c'est l'objet de la section suivante) ils restent en mémoire (en session).

7.2 Affichage des messages flash

Travail à effectuer

On écrit maintenant une action avec sa vue qui affiche les messages flash.
Il suffit de suivre les indications au fur et à mesure pour avoir un code effectif.

L'action sert uniquement à afficher la vue.

Vous pouvez récupérer le code complet du contrôleur à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8dabe433daea693ef3c9c00ed40c600987bf5120/src/Controller/Sandbox/InjectionController.php

Et voici directement le code de l'action ajoutée :

Sandbox/InjectionController.php (extrait)

```
48      #[Route('/display-flash', name: '_display_flash')]
49      public function displayFlashAction(): Response
50      {
51          return $this->render('Sandbox/Injection/displayFlash.html.twig');
52      }
```

Explications : l'action n'a pas pour rôle de faire des affichages, tout le code sera dans la vue.

Si l'interception des redirections est active (ce qui peut être vraiment utile en phase de développement), avant de déclencher la redirection, rappelez-vous que vous pouvez cliquer sur la toolbar et choisir l'entrée "Request/Response" et visualiser l'onglet dédié aux messages flash.

Et enfin la vue qui se charge d'afficher les messages flash.

Vous pouvez récupérer le code complet du template à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/8dabe433daea693ef3c9c00ed40c600987bf5120/templates/Sandbox/Injection/displayFlash.html.twig

Et voici directement le code de la vue :

templates/Sandbox/Injection/displayFlash.html.twig (extrait)

```
8       <h1>Messages flash de type "info"</h1>
9       <ul>
10          {% for msg in app.session.flashBag.get('info') %}
11              <li>{{ msg }}</li>
12          {% endfor %}
13      </ul>
```

Explications :

- *ligne 10* : la variable *app* dans Twig est globale et contient, entre autre, la session (référez-vous à la documentation officielle pour un aperçu complet de cette variable).
- *ligne 10* : la méthode *get* de l'objet *flashBag* permet de récupérer l'ensemble des messages d'une catégorie.

- le fait d’afficher les messages les fait disparaître de la session : si on rafraîchit la page, les messages ne se réaffichent pas.

Complétez le code en ajoutant deux messages flash de type ’error’ dans l’action *createFlash*. Ces nouveaux messages doivent s’afficher dans le Twig de l’action *displayFlash*.

Vous pouvez récupérer les codes complets à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/a8c024d9c286f6c13e8be3a1455bba31442cb442/src/Controller/Sandbox/InjectionController.php

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/a8c024d9c286f6c13e8be3a1455bba31442cb442/templates/Sandbox/Injection/displayFlash.html.twig

Ce qui donne :

Sandbox/InjectionController.php (extrait)

```
39      #[Route('/create-flash', name: '_create_flash')]
40      public function createFlashAction(Session $session): Response
41      {
42          // par exemple cette action supprime une entrée dans la base de données
43          $session->getFlashBag()->add('info', 'L\'enregistrement a été supprimé');
44          $this->addFlash('info', 'L\'enregistrement a été supprimé (bis repetita)');
45          $this->addFlash('error', 'Il pourrait y avoir une erreur');
46          $this->addFlash('error', 'Il pourrait y avoir une erreur (bis repetita)');
47          return $this->redirectToRoute('sandbox_injection_display_flash');
48      }
```

templates/Sandbox/Injection/displayFlash.html.twig (extrait)

```
8           <h1>Messages flash de type "info"</h1>
9           <ul>
10              {% for msg in app.session.flashBag.get('info') %}
11                  <li>{{ msg }}</li>
12              {% endfor %}
13          </ul>
14
15          <h1>Messages flash de type "error"</h1>
16          <ul>
17              {% for msg in app.session.flashBag.get('error') %}
18                  <li>{{ msg }}</li>
19              {% endfor %}
20          </ul>
```

8 Application “vente en ligne”

On travaille désormais sur la “vraie” application dans les contrôleurs autres que *Sandbox* (qui ne servent qu’à s’entraîner).

Pour chaque route demandée, il faut fournir une action et une vue dédiées (si une vue a un sens). La vue affichera un titre représentatif et les paramètres reçus.

On rappelle qu’il n’est pas demandé d’interroger la base de données, mais uniquement préparer les routes, actions et vues qui le feront ultérieurement.

8.1 Base de données

Travail à effectuer

Il s’agit juste de comprendre la base de données.

La base de données vous est fournie et contient 7 tables, dont deux de jointure :

- produits
- manuels
- images
- pays
- magasins
- produits_pays

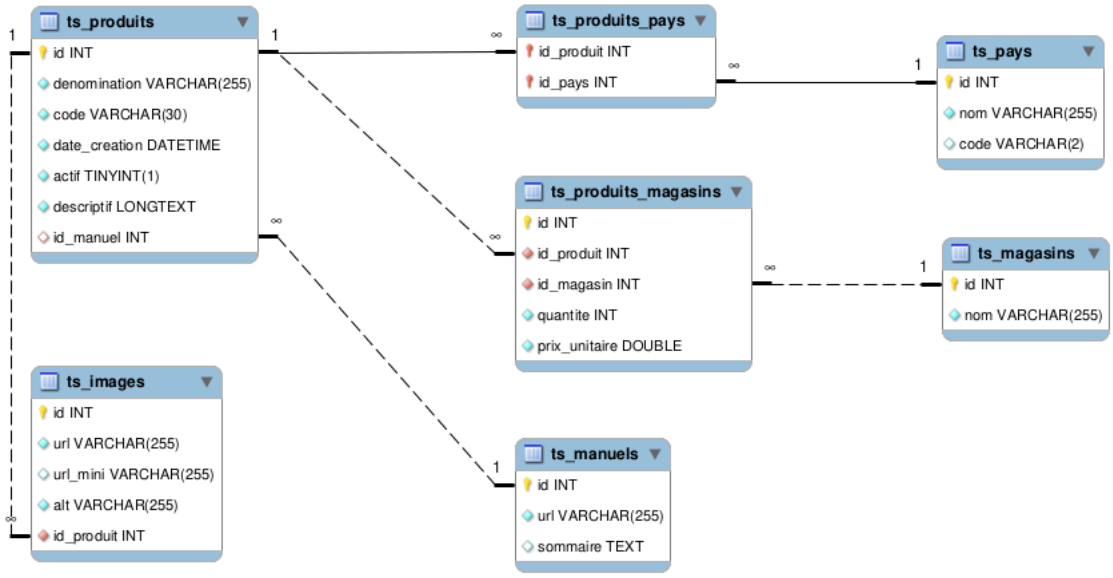
- produits_magasins

Un produit peut ou non avoir un manuel d'utilisation ; et un manuel est toujours lié à un et un seul produit.

Un produit peut avoir plusieurs images, mais une image est liée à un et un seul produit.

Un produit peut être vendu dans plusieurs pays ; et un pays peut vendre plusieurs produits.

On a une même relation (i.e. *n-n*) entre les produits et les magasins. La différence est que cette relation contient deux attributs.



Référez-vous au code SQL pour plus de précisions.
On insiste sur le fait que ce code SQL n'est fourni qu'à titre de documentation et ne sera pas utilisé techniquement.

8.2 CRUD sur la table produit

CRUD : Create, Read, Update, Delete.

Travail à effectuer

Le but de faire le squelette du contrôleur qui gère la table *produits* avec les vues correspondantes.

Créez le contrôleur *Produit*.
Utilisez "/produit" comme préfixe à toutes les routes de ce contrôleur.

On demande les actions suivantes :

- *listAction* : liste tous les produits
  - URL : /produit/list/xxx
  - *xxx* est un nombre strictement positif (à contraindre dans les annotations)
  - le paramètre *xxx* est un numéro de page facultatif (0 par défaut, cf. ci-dessous pour les explications)
  - trouvez un nom plus adéquat que *xxx* pour le paramètre [11].
  - le contrôleur crée un tableau de 3 produits en dur (pour simuler une requête à la base de données) et l'envoie à la vue Twig.
  - la vue Twig affiche les messages flash de type "info"
  - la vue Twig affiche le numéro de la page
  - la vue Twing affiche les produits, mais uniquement la dénomination et le code barre. S'il vous manque trop de connaissances, passez cette étape.

11. spoiler : *page* par exemple

    - terminez la vue par un affichage, avec la fonction *dump*, du tableau de produits.
- *indexAction* : accueil du contrôleur *Produit*
    - URL : /produit
    - il s'agit uniquement d'une redirection vers la route "list" avec le paramètre 1
    - habituellement cette méthode est la première de la classe
- *viewAction* : affiche le détail d'un produit
    - URL : /produit/view/xxx
    - le paramètre *xxx* est un *id* (clé primaire dans la base de données)
    - le paramètre est obligatoire (pas de valeur par défaut)
    - *xxx* est un nombre strictement positif
    - trouvez un nom plus adéquat que *xxx* pour le paramètre [12].
    - le contrôleur "crée" un produit en dur et l'envoie à la vue Twig.
    - la vue Twig affiche les messages flash de type "info" (nous verrons plus tard comment ne pas dupliquer ce code)
    - la vue Twig affiche le produit avec la fonction *dump*.
- *addAction* : ajout d'un produit dans la base de données
    - URL : /produit/add
    - pas de paramètre
    - à terme il y aura un formulaire et le traitement de celui-ci.
    - créez un message flash de type "info" indiquant la réussite (ou plutôt l'échec) de l'ajout.
    - pour l'instant on fait une redirection vers la route "view" avec le paramètre 3 (ou un autre à votre convenance)
- *editAction* : modification d'un produit existant
    - URL : /produit/edit/xxx
    - le paramètre *xxx* est l'*id* (clé primaire dans la base de données) d'un produit existant
    - le paramètre est obligatoire (pas de valeur par défaut)
    - *xxx* est un nombre strictement positif
    - trouvez un nom plus adéquat que *xxx* pour le paramètre [13].
    - à terme il y aura un formulaire et le traitement de celui-ci.
    - créez un message flash de type "info" indiquant la réussite (ou plutôt l'échec) de la modification.
    - pour l'instant on fait une redirection vers la route "view" avec le paramètre *xxx*
- *deleteAction* : suppression d'un produit
    - URL : /produit/delete/xxx
    - le paramètre *xxx* est un *id* (clé primaire dans la base de données)
    - le paramètre est obligatoire (pas de valeur par défaut)
    - *xxx* est un nombre strictement positif
    - trouvez un nom plus adéquat que *xxx* pour le paramètre.
    - créez un message flash de type "info" indiquant la réussite (ou plutôt l'échec) de la suppression.
    - pour l'instant on fait une redirection vers la route "list" sans paramètre

L'action *list* a donc comme paramètre le numéro de la page à afficher (si par exemple les enregistrements sont affichés 10 par 10). Si ce numéro est strictement positif alors on affiche la page correspondante : s'il vaut 0 alors on affiche tous les enregistrements sans pagination. Ces explications sont présentes à titre indicatif mais n'ont pas à être gérées dans cet exercice.
Au niveau de la route le paramètre est facultatif. S'il est fourni il doit être strictement positif ; s'il n'est pas fourni il vaudra alors 0 (c'est un cas où la valeur par défaut ne respecte pas les restrictions du paramètre).

Vous avez une propostion de code à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/08e1177327ca0403ddae2d0da9a20fb61b5b588d

12. re-spoiler : *id* par exemple
13. pas de spoil cette fois-ci

8.3 CRUD et les tables principales

Travail à effectuer

Aucun code n'est à produire dans cette section.

Il faudrait les contrôleurs *Manuel*, *Image*, *Pays* et *Magasin* pour les tables correspondantes, avec les mêmes actions que le contrôleur *Produit*.

Pour l'instant on laisse de côté ces contrôleurs et on ne fait rien dans cet exercice.

8.4 CRUD et jointures

Travail à effectuer

Le but est de créer les squelettes de deux actions pour alimenter les tables de jointure.

Créez deux routes :

- une pour ajouter une relation produit/pays
- une pour ajouter une relation produit/magasin

La question est : dans quel(s) contrôleur(s) écrire ces actions ?

Créer un contrôleur pour les tables de jointure est un peu violent : nous faisons le choix de rattacher ces actions au contrôleur de la table "produits" : *ProduitController*.

Ces deux actions ne prennent pas de paramètre : il y aura à terme un formulaire. Pour l'instant elles ont le même modèle que l'action "add" du contrôleur *Produit*.

Par exemple les routes pourraient être :

- /produit/pays/add
- /produit/magasin/add

Pensez à mettre un message flash dans chaque action.

Vous avez une propostion de code à :

https://gitlab.com/subrenat/l3-web-2022-23/-/blob/7d2798b9ddb2eb95c9567e7521ac4f3624cca290/src/Controller/ProduitController.php

8.5 Autres actions

Travail à effectuer

Le but est de créer deux actions spécifiques à la table *magasins*.

Tous comptes faits, créez le contrôleur *Magasin*.
Utilisez "/magasin" comme préfixe à toutes les routes de ce contrôleur.

Préparez une action, pour un magasin, qui fera un récapitulatif de la valeur totale des produits en stock, i.e. $\sum_{produis} qte_i * prix_i$. Cette action calcule donc une seule valeur.

- La route pourrait être : /magasin/valeur-stock/{id}
- Le paramètre, obligatoire, est la clé primaire désignant le magasin.
- En attendant de pouvoir interroger la base de données, l'action fournira une valeur fictive en dur à la vue.
- La vue se contentera d'afficher ce nombre.

Enfin faites une action, toujours pour un magasin, qui affiche la liste des produits dont le prix unitaire est compris dans une fourchette.

- La route pourrait être : /magasin/stock/{id}/{valinf}/{valsup}
- Le premier paramètre, obligatoire, est la clé primaire désignant le magasin.
- Le second, facultatif, est la valeur inférieure de la fourchette. C'est un entier positif dont la valeur par défaut est 0.

- Le troisième, facultatif, est la valeur supérieure de la fourchette. C'est soit -1, soit entier un positif et la valeur par défaut est -1 (qui signifie que la fourchette n'a pas de borne supérieure).
- En attendant de pouvoir interroger la base de données, l'action fournira une liste fictive en dur de triplets (nom du produit, prix unitaire, quantité en stock).
- La vue affichera, dans un tableau HTML, non seulement les données fournies par l'action, mais également, pour chaque produit, la valeur totale (prix unitaire multiplié par la quantité). En outre elle affichera le nombre total de produits et la valeur totale du stock (uniquement pour les produits affichés).
  L'action se contente de passer les données à la vue, et c'est cette dernière qui fait tous les calculs.

9 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/TP3

Les différentes étapes du TP sont dans la branche *b-TP3*.

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