

Suivez scrupuleusement les instructions ci-dessous. Lisez bien ce qui s'affiche et n'hésitez pas à appeler l'enseignant si l'installation semble avoir échoué.

1.1 – Installation du projet Symfony fourni

- **Prérequis : vérifiez que vous disposez bien d'une centaine de Mo disponible dans votre quota disque. Sinon, faites du ménage ou vous ne pourrez pas installer correctement votre projet symfony !**

- Placez-vous dans le répertoire qui contient le projet symfony (version 7.2) qui a déjà été créé pour vous (remplacez **votre_login** par votre login UGA habituel) :

```
cd /users/info/pub/2a/R4.01/TP/votre_login
```

C'est dans ce répertoire que vous devez, au fil des séances, développer votre projet. Ne le déplacez pas ! C'est ici que nous viendrons contrôler et évaluer le travail que vous aurez réalisé en TP.

- Depuis votre répertoire, lancez la commande d'installation des dépendances :

```
composer install
```

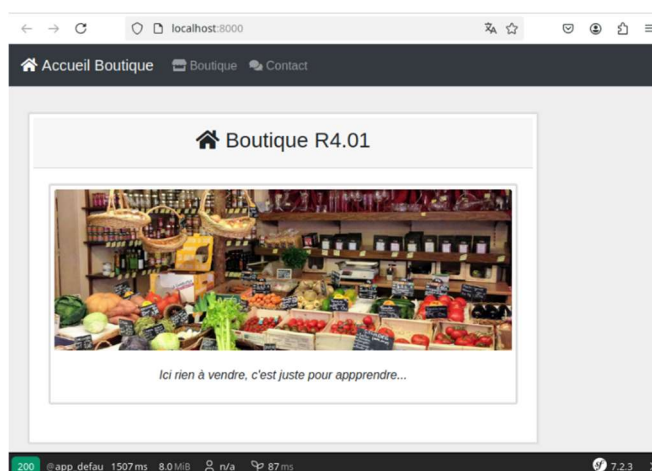
- Lancez l'IDE **PhpStorm** avec lequel nous travaillerons. C'est le meilleur pour développer en Symfony, en particulier pour avoir une coloration syntaxique et un formatage du code justes dans les *templates Twig* (vous constaterez par la suite que c'est très utile) :

```
phpstorm &
```

- Ouvrez votre projet : `/users/info/pub/2a/R4.01/TP/votre_login`
- Depuis **PhpStorm**, lancez le serveur de développement de php qui fera tourner notre projet. Ce serveur permet de déboguer le code PHP grâce au débogueur très intuitif de PhpStorm (même fonctionnement que sous Clion ou IntelliJ). Cliquez sur la flèche verte :



- Lancez **firefox** et naviguez vers l'URL <http://localhost:8000/>
- Si tout s'est bien passé, vous devriez voir dans votre navigateur la page d'accueil par défaut du squelette de projet Symfony fourni :



Travail à réaliser au fil des TP : site de e-commerce

Le site devra offrir un *front-office* permettant à des utilisateurs de :

- Consulter le catalogue d'un commerçant : catégories de produits, liste des produits d'une catégorie donnée, caractéristiques d'un produit (libellé, photo, prix, etc...).
 - Constituer un « panier » et passer commande en s'inscrivant sur le site.
 - Revenir ensuite consulter l'état d'avancement de leurs commandes.
- Une démo du site à réaliser est visible :
<https://iut2-dg-scolarité.iut2.univ-grenoble-alpes.fr/info/R401/>

Nota Bene : L'objectif de ce module n'est pas de consacrer son temps à faire du « design » avec Bootstrap 4 (le framework css utilisé pour ce projet). Vous pouvez donc vous inspirer sans vergogne du code HTML du site de démo dans vos templates Twig

1.2 - Mise en place du Layout et de sa barre de navigation

- Un exemple de fichier **templates/base.html.twig**, déjà fourni, contient le *layout* des pages de votre site. Ce *layout* « factorise » le code HTML commun à vos différentes pages et définit des blocs qui doivent ensuite être redéfinis dans les *templates* qui hériteront de lui. Sur notre exemple, on propose :
 - Un bloc **title** pour pouvoir remplir la balise <title> dans chaque *template* fils
 - Un bloc **body** pour pouvoir remplir la <div> qui héberge le contenu principal d'un *template* fils
 - Un bloc **script** pour pouvoir ajouter, à chaque *template* fils, les éventuels scripts js dont il aurait spécifiquement besoin
 - Un bloc **style** pour pouvoir ajouter, à chaque *template* fils, les éventuels styles css dont il aurait spécifiquement besoinOn pourrait bien sûr définir d'autres blocs !
- Par souci de modularité, la barre de navigation n'est pas codée dans **base.html.twig** mais dans un autre *template*, **templates/navbar.html.twig**. Cette barre de navigation est incluse dans le *layout* grâce à une balise Twig `{% include ... %}`
- Au fur et à mesure que vous créerez les différentes pages de votre site, des liens vers les nouvelles pages devront être correctement mis en place dans la barre de navigation en utilisant des noms de routes `{{ path("un_nom_de_route", ...) }}`

1.3 - Pages statiques : créer la page contact

- Un squelette de contrôleur **DefaultController** a déjà été créé avec la commande :
bin/console make:controller (ne la relancez pas pour l'instant !)
- Cette commande a produit 2 fichiers :
 - Une classe **DefaultController.php** dans le répertoire **src/Controller/**, qui contient déjà une méthode **index** avec sa route associée (en annotation)
 - Un *template* **index.html.twig** dans le répertoire **templates/default**
- Dans **src/Controller/DefaultController.php**, la route **app_default** est définie pour que la méthode **index** soit déclenchée par l'URL /
- Le fichier **templates/default/index.html.twig** affiche l'image de la page d'accueil de votre site (en héritant de **base.html.twig**)
- Dans **src/Controller/DefaultController.php**, créez une nouvelle méthode **contact** avec une route nommée **app_default_contact** associée à l'URL **/contact**

- La méthode **contact** devra rendre le *template* `contact.html.twig` que vous créerez dans le répertoire **templates/default/** et qui devra afficher (en héritant lui aussi de **base.html.twig**) la page de contact de votre site.
- Faites en sorte que, dans la barre de navigation, le lien contact du menu permette de naviguer vers cette nouvelle page.

1.4 - Pages dynamiques : catégories et produits par catégorie

Nous n'avons pas encore modélisé les données persistantes de notre application (ce sera au TP4). En attendant, pour pouvoir réaliser une maquette de la partie « boutique » de notre site, on vous fournit un service Symfony (une classe PHP) qui permettra de manipuler des **Catégories** de produits et des **Produits** (stockés « en dur » dans des tableaux PHP). L'interface de ce service, que vous devrez utiliser, est la suivante :

```
// Un service pour manipuler le contenu de la Boutique
// qui est composée de catégories et de produits stockés "en dur"
class BoutiqueService {
    // renvoie toutes les catégories
    public function findAllCategories() : array {...}
    // renvoie la catégorie dont id == $idCategorie (null si pas trouvée)
    public function findCategorieById(int $idCategorie) : object {...}
    // renvoie le produit dont id == $idProduit, null si pas trouvé
    public function findProduitById(int $idProduit) : object {...}
    // renvoie un tableau de produits dont idCategorie == $idCategorie
    public function findProduitsByCategorie(int $idCategorie) : array {...}
    // renvoie un tableau de produits dont libelle+texte contient $search
    public function findProduitsByLibelleOrTexte(string $search) : array {...}
    // Le catalogue de la boutique, codé en dur dans un tableau associatif
```

Les méthodes fournissent :

- Soit un objet standard PHP qui représente un **Produit** ou une **Catégorie**
- Soit un tableau PHP d'objets standards de type **Produit** ou de **Catégorie**

Ces objets sont construits à partir de données stockées « en dur », au format JSON, dans le service lui-même. Observez bien les différents attributs qui sont disponibles dans ces 2 types d'objet :

```
private $categories = <<<JSON
[
    {
        "id"      : 1,
        "libelle" : "Fruits",
        "visuel"  : "images/categories/fruits.jpg",
        "texte"   : "De la passion ou de ton imagination"
    }, ...
]
JSON;
private $produits = <<<JSON
[
    {
        "id"           : 1,
        "idCategorie"  : 1,
        "libelle"       : "Pomme",
        "texte"         : "Elle est bonne pour la tienne",
        "visuel"        : "images/produits/pommes.jpg",
        "prix"          : 3.42
    }, ...
]
JSON;
}
```

- Le fichier **BoutiqueService.php** est disponible dans le répertoire **src/Service**

Lorsque vous voudrez utiliser ce service dans une méthode d'un contrôleur, il vous suffira de l'injecter par les paramètres du contrôleur. Par exemple :

```
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Attribute\Route;
use App\Service\BoutiqueService;

class BoutiqueController extends AbstractController {

    #[Route('/boutique', name : 'app_boutique')]
    public function index(BoutiqueService $boutique) : Response {

        // Utiliser le service pour récupérer les catégories
        $categories = $boutique->findAllCategories();

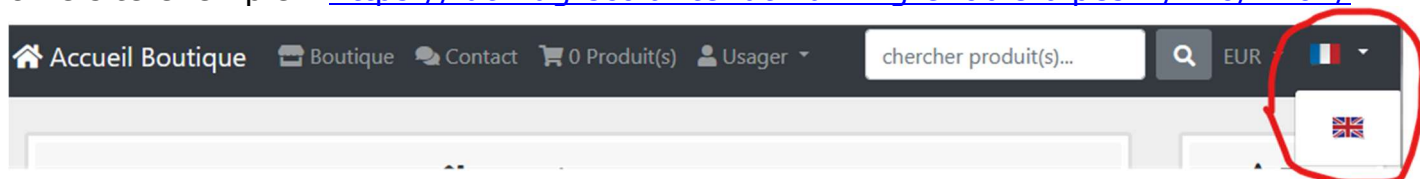
        // Rendre un template auquel on transmet ces catégories
        // ...
    }
}
```

- Avec la commande **bin/console make:controller** évoquée précédemment, créez un nouveau contrôleur **BoutiqueController** qui sera chargé d'afficher la page des « rayons » (toutes les catégories disponibles) et la page de détail d'un rayon (tous les produits d'une catégorie donnée)
- Définissez une route **app_boutique** pour l'URL **/boutique**. Cette route exécutera la méthode **index** du contrôleur **BoutiqueController** et devra afficher la liste des **catégories** de notre boutique.
- Définissez une route **app_boutique_rayon** pour l'URL **/rayon/{idCategorie}**. Cette route exécutera la méthode **rayon(int \$idCategorie)** du contrôleur **BoutiqueController** et devra afficher la liste des **produits** de la catégorie **idCategorie**.
- Ecrivez le *template* **templates/boutique/index.html.twig** qui affichera les catégories, chaque catégorie étant un lien pour naviguer vers les produits de cette catégorie
- Ecrivez le *template* **templates/boutique/rayon.html.twig** qui affiche les produits d'une catégorie
- Complétez la barre de navigation pour pouvoir atteindre la page **app_boutique**

👤 **Prérequis** : Vous devez impérativement terminer le TP01 avant de débiter celui-ci

2.0 - Objectif du TP

L'objectif de ce TP est de mettre en place l'internationalisation (i18n) sur notre embryon d'application Symfony. Nous allons faire en sorte que les 2 pages statiques, créées lors du TP01, puissent être consultées aussi bien en français qu'en anglais (et/ou toute autre langue qu'il vous plaira d'ajouter). Cela sous-entend bien sûr que le contenu de la barre de navigation devra aussi être traduit. Cette même barre de navigation devra également comporter un petit menu déroulant permettant de choisir la langue affichée sur le site, cf le site exemple : <https://iut2-dg-scolarité.iut2.univ-grenoble-alpes.fr/info/R401/>



⚠️ *Lors des prochains TP, nous ne nous « amuserons » plus à traduire les contenus des nouveaux templates que nous créerons. **Par contre, nous devons bien faire en sorte que toutes les routes créées aient toujours des URL préfixées par la locale.***

2.1 – Internationaliser

Les différentes étapes à réaliser vous ont été décrites dans le cours n°2. Il vous suffit donc de suivre ces étapes (en étant vigilant·e). ⚠️ *Lorsque vous modifiez un fichier de configuration au format YAML, soyez vigilant avec l'indentation. Les informations de même niveau doivent avoir la même indentation, à l'espace près !*

Rappel des étapes à réaliser :

1. Définissez la **locale par défaut** et la **locale de repli** dans le fichier `config/packages/translation.yaml` (cf cours 02, page 4).
2. Définissez les **locales supportées** par votre application dans le fichier `config/services.yaml` (cf cours 02, page 5).
3. Modifiez **toutes les routes** de votre application pour que la locale soit transmise dans chaque URL. Pour cela, l'URL doit être préfixée par un paramètre `{_locale}` dont on contraindra les valeurs possibles avec un attribut **requirements** qui sera égal à l'expression régulière qui vérifie la liste des locales supportées. Pour la route de la page d'accueil, on rajoutera aussi un attribut **defaults** pour donner une valeur par défaut à la locale lorsque l'on navigue pour la première fois vers la page d'accueil du site. (cf cours 02, page 4-5).
4. Modifiez **tous vos templates** (y compris le *layout* et la *navbar*) en encadrant les éléments (mot, phrase ou paragraphe) que vous voulez traduire par des balises Twig `{% trans %}...{% endtrans %}`. Les éléments à traduire devront être remplacés par un identifiant de la forme : `repertoire_du_template.nom_du_template.identifiant_element_a_traduire`

Ces identifiants seront donc ceux qui apparaîtront comme **source** dans les catalogues qui seront produits à l'étape suivante (cf cours 02, page 8).

5. Faire produire au CLI de symfony, pour chaque langue **LL** proposée (donc au moins **fr** et **en**), le catalogue qui contiendra toutes les sources définies à l'étape précédente. Pour la langue **LL**, utilisez la commande :

```
php bin/console translation:extract --force LL (cf cours 02, page 11).
```

6. L'étape précédente a produit, pour chaque locale **LL**, un catalogue (fichier XLIFF) nommé **translations/messages+intl-icu.LL.xlf**. Vous devez maintenant jouer au traducteur ! Dans chacun de ces catalogues, remplissez les balises **<target>**, associées à chaque balise **<source>**, afin d'indiquer par quoi sera remplacé l'identifiant défini dans la source lorsqu'il sera affiché avec la locale **LL** (cf cours 02, page 10).

Les étapes 4,5,6 peuvent être répétées plusieurs fois sans perdre le travail qui a été fait précédemment.

Pour vérifier que la traduction est bien en place, naviguez vers les pages qui sont censées être traduites et vérifiez que la locale est prise en compte :

- <http://localhost:8000/fr> doit afficher votre page d'accueil en français
- <http://localhost:8000/en> doit afficher cette même page, mais en anglais

... et ceci doit fonctionner pour toutes pages de votre site !

2.2 – Menu « langue » dans la barre de navigation

On souhaite maintenant disposer dans la barre de navigation (affichée sur chaque page du site) d'un menu déroulant permettant à l'utilisateur de choisir la langue dans laquelle est affichée la page actuelle. Il faut donc coder ce menu déroulant, à l'aide de Twig, dans le *template navbar.html.twig*.

Quelques indications pour réaliser cela (lisez-les toutes avant de coder !) :

1. Il faut avoir accès dans le *template* à la liste des locales supportées par l'application. Ce paramètre, défini lors de l'étape 2, doit être « injecté » dans Twig en complétant le fichier **config/packages/twig.yml** de la façon suivante :

```
#config/packages/twig.yml
```

```
twig:
```

```
  globals:
```

```
    supported_locales: '%app.supported_locales%'
```

2. Vous disposerez alors, dans vos *templates* Twig, d'une variable **supported_locales** qui sera une chaîne de caractères ('**fr|en**') contenant les locales supportées, séparées par le caractère '|'. Pour parcourir ces locales dans une itération, vous pourrez utiliser la fonction **split** de Twig qui permet de convertir une chaîne de caractères en tableau, en précisant quel est le séparateur des différents éléments :
{% for uneLocale in supported_locales | split('|') %} ... {% endfor %}

3. Le titre du menu déroulant doit être un texte (ou une image) correspondant à la locale actuelle. Pour savoir quelle est la valeur actuelle de la locale, vous pouvez en Twig accéder à la requête HTTP et y récupérer la locale actuelle :
{% set locale = app.request.attributes.get('_locale') %}

4. Dans l'itération qui va produire le contenu du menu déroulant, il faudra pour chaque locale supportée (sauf la locale actuelle !) produire une URL permettant de naviguer vers la page actuelle, mais en changeant la locale définie dans l'URL de cette page. Rappelez-vous qu'en Symfony+Twig, une URL ne doit pas être « codée » en dur mais doit être forgée, à partir d'un nom de route, à l'aide de la fonction Twig **path**. Si la route en question contient des paramètres, ceux-ci doivent être fournis à la fonction **path** dans un tableau associatif Twig : `{'param1': 'val. param1', ...}`. Pour pouvoir fabriquer nos liens de rechargement de la page actuelle, il va falloir :

4.a - Disposer du nom de la route actuelle (**route**) :

```
{% set route = app.request.attributes.get('_route') %}
```

4.b - Disposer du tableau des paramètres de la route actuelle (**params**) :

```
{% set params = app.request.attributes.get('_route_params') %}
```

4.c - Modifier la valeur du paramètre `'_locale'` dans **params** en lui donnant la valeur de la locale (**uneLocale**) dans laquelle on veut afficher la page :

```
{% set params = params | merge({'_locale': uneLocale }) %}
```

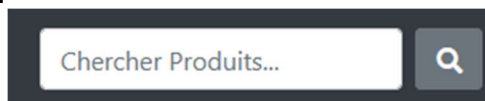
4.d - Et enfin forger l'URL pour naviguer vers la page actuelle, en utilisant la route actuelle et ses paramètres modifiés :

```
<a href="{ path(route,params) }" ...>
```

Vous avez bien lu ces indications ? Bravo, vous êtes prêt·e à coder le menu déroulant !

2.3 – Champ de recherche dans la barre de navigation

Il vous reste maintenant à coder le champ de recherche de produits disponible lui aussi dans la barre de navigation :



Quelques indications rapides pour faire cela :

1. Dans le contrôleur **BoutiqueController**, coder une méthode et sa route (**app_boutique_chercher**) qui pourra répondre à une recherche de produits.

- L'URL de la route devra avoir un paramètre **recherche** qui contiendra la chaîne de caractères à rechercher parmi les attributs **libelle** et **texte** de nos produits :

`/_{locale}/boutique/chercher/{recherche}`

⚠ Il faut, dans la route, donner par défaut la valeur chaîne vide ("") au paramètre **recherche** (*vous comprendrez bientôt pourquoi...*) :

```
#[Route(
    path: '/chercher/{recherche}',
    name: 'app_boutique_chercher',
    requirements: ['recherche' => '.*'], // regexp pour avoir tous les car, / compris
    defaults: ['recherche' => ''])]
```

- La méthode associée à la route aura le prototype suivant :

```
public function chercher(BoutiqueService $boutique,
                        string $recherche) : Response
```

Cette méthode devra utiliser **BoutiqueService** pour trouver la liste des produits dont **libelle** ou **texte** contiennent **\$recherche** (voir la méthode proposée dans le service : **findProduitsByLibelleOrTexte**).

Le résultat sera alors transmis à un *template* **boutique/chercher.html.twig** qui se chargera d'afficher le résultat de la recherche.

2. Dans le *template navbar.html.twig*, il faudra disposer d'un champ **<input>** pour pouvoir saisir la recherche et d'un bouton qui, lorsqu'il sera cliqué, devra déclencher une navigation vers l'URL de la page de recherche, URL à laquelle il faudra, grâce à un peu de **JavaScript**, concaténer la valeur du champ **<input>**.
Cette URL devra en partie être forgée en Twig avec la fonction **path** (*et c'est à ce moment que, Eurêka !, vous devez comprendre pourquoi le paramètre recherche de la route doit avoir une valeur par défaut à chaîne vide*).
3. Réfléchissez au meilleur endroit possible pour écrire votre code JS qui est censé se trouver sur toutes les pages de votre site.
4. Pour que tout fonctionne sans problème, même quand la chaîne de recherche contient des caractères spéciaux qui ont un sens particulier dans une URL, il faudra :
 - Côté **JS** : encoder la chaîne de recherche, avant de forger l'URL, en utilisant la fonction **encodeURIComponent**
 - Côté fonction **chercher** en **PHP** : décoder la chaîne de recherche reçue en utilisant la fonction **urldecode**

3.0. Travail à réaliser – Vue d'ensemble

Pour mettre en place la notion de « panier » sur notre site, nous allons :

- Développer une nouvelle **fonctionnalité métier**, le panier, dans un service **PanierService** qui devra :
 - Proposer les opérations de base à réaliser sur un panier :
 - Ajouter/Enlever un produit au/du panier
 - Supprimer un produit du panier
 - Calculer le montant du panier, le nombre de produits dans le panier
 - Vider tout le panier
 - Consulter le contenu du panier
 - Assurer la persistance du panier de l'utilisateur en stockant son contenu (identifiants des Produits concernés et quantité) en **session**
- Mettre en place les routes, contrôleurs et le *template* nécessaires pour offrir à l'utilisateur une interface qui lui permettra d'interagir avec ce service
- Exemple : <https://iut2-dg-scolarité.iut2.univ-grenoble-alpes.fr/info/R401/>

3.1. Développez le service PanierService

Le service **PanierService** va devoir utiliser le service **BoutiqueService** afin d'accéder aux Produits. Il va également devoir aussi utiliser la **Session** afin de pouvoir stocker le contenu du panier en session.

On vous fournit ci-dessous l'interface de ce nouveau service qu'il vous faudra compléter (téléchargez le fichier PanierService.php sur Chamilo) :

```
<?php // src/Service/PanierService.php
namespace App\Service;
use Symfony\Component\HttpFoundation\RequestStack;
use App\Service\BoutiqueService;
// Service pour manipuler le panier et le stocker en session
class PanierService {
    private $session; // Le service session
    private $boutique; // Le service boutique
    private $panier; // Tableau associatif, la clé est un idProduit, la valeur une quantité
                    // $this->panier[$idProduit] = quantité du produit $idProduit
    const PANIER_SESSION = 'panier'; // Nom de la var. de session pour persister $this->panier
    // Constructeur du service
    public function __construct(RequestStack $requestStack, BoutiqueService $boutique) {
        // Récupération des services session et BoutiqueService
        $this->boutique = $boutique;
        $this->session = $requestStack->getSession();
        // Récupération du panier en session s'il existe, initialisation à tableau vide sinon
        $this->panier = // à compléter... ;
    }
    // Ajouter au panier le produit $idProduit en quantité $quantite
    public function ajouterProduit(int $idProduit, int $quantite = 1) : void { // à compléter... }
    // Enlever du panier le produit $idProduit en quantité $quantite
    public function enleverProduit(int $idProduit, int $quantite = 1) : void { // à compléter... }
    // Supprimer le produit $idProduit du panier
    public function supprimerProduit(int $idProduit) : void { // à compléter... }
    // Renvoyer le montant total du panier
    public function getTotal() : float { // à compléter... }
    // Renvoyer le nombre de produits dans le panier
    public function getNombreProduits() : int { // à compléter... }
    // Vider complètement le panier
    public function vider() : void { // à compléter... }
    // Renvoyer le contenu du panier (dans le but de l'afficher dans un template)
    // => un tableau d'éléments [ "produit" => un objet produit, "quantite" => sa quantite ]
    public function getContenu() : array { // à compléter... }
}
```

Lisez les indications page suivante avant de vous lancer dans le code !

🔗 Quelques indications utiles :





- Rappels sur la manipulation des tableaux en PHP... Soit `$tableau` un tableau PHP :
 - `foreach($tableau as $cle => $val)` est l'itération qui permet de parcourir `$tableau`
 - `isset($tableau[$i])` renvoie vrai si l'élément d'indice `$i` est bien présent dans `$tableau`, faux sinon
 - `unset($tableau[$i])` supprime l'élément d'indice `$i` de `$tableau`
 - `$tableau = []` vide le contenu de `$tableau`
- Pour l'utilisation de la **session** en symfony, voir **cours 03, page 11**.

3.2. Routes, Contrôleurs, Template

Dans le répertoire **src/Controller**, créez une nouvelle classe contrôleur appelé **PanierController** (en utilisant la commande **php bin/console make:controller**)

Cette classe contiendra tous les contrôleurs (méthodes) permettant à l'utilisateur d'interagir avec son panier *via* des requêtes HTTP. Ces contrôleurs devront bien sûr utiliser les services **PanierService** et **BoutiqueService**. Vous ferez en sorte de lever des exceptions si une requête n'a pas de sens, par exemple si on essaye d'ajouter au panier un produit dont l'identifiant n'existe pas (et pour déterminer cela, il faudra utiliser le service **BoutiqueService**). Pour lever une exception et renvoyer une réponse http 404, voir **cours 03 page 10**.

Dans cette classe contrôleur, vous devrez proposer les routes et contrôleurs (méthodes) suivants :

- Une route **app_panier_index** pour l'URL `/_{locale}/panier/`
 - Cette route sera associée à la méthode **index**
 - La méthode **index** devra utiliser le service **PanierService** pour récupérer le contenu du panier et le transmettre au *template* **panier/index.html.twig**
 - Ce *template* devra afficher le contenu du panier qui lui a été transmis
- Une route **app_panier_ajouter** pour l'URL `/_{locale}/panier/ajouter/{idProduit}/{quantite}`
 - Cette route sera associée à la méthode **ajouter** et devra utiliser le service **PanierService** pour ajouter au panier le produit reçu en paramètre, s'il existe (dans la quantité précisée).
 - La méthode **ajouter** devra **rediriger** l'utilisateur vers la route **panier_index** (voir **cours 03 page 7**)
 - Modifiez le *template* qui affiche les produits (**boutique/rayon.html.twig**) afin qu'un clic sur un produit déclenche la route **app_panier_ajouter**
 - Modifiez le *template* **panier/index.html.twig** pour offrir sur chaque produit du panier un bouton  qui déclenchera la route **app_panier_ajouter**
- Une route **app_panier_enlever** pour l'URL `/_{locale}/panier/enlever/{idProduit}/{quantite}`
 - Cette route sera associée à la méthode **enlever** et devra utiliser le service **PanierService** pour enlever au panier le produit reçu en paramètre, s'il existe (dans la quantité précisée).
 - La méthode **enlever** devra rediriger l'utilisateur vers la route **app_panier_index**
 - Modifiez le *template* **panier/index.html.twig** pour offrir sur chaque produit du panier un bouton  qui déclenchera la route **app_panier_enlever**
- Une route **app_panier_supprimer** pour l'URL `/_{locale}/panier/supprimer/{idProduit}`
 - Cette route sera associée à la méthode **supprimer** et devra utiliser le service **PanierService** pour enlever au panier le produit reçu en paramètre, s'il existe.
 - L'action **supprimer** devra rediriger l'utilisateur vers la route **panier_index**
 - Modifiez le *template* **panier/index.html.twig** pour offrir sur chaque produit du panier un bouton  qui déclenchera la route **app_panier_supprimer**
- Définissez une route **app_panier_vider** pour l'URL `/_{locale}/panier/vider`
 - Cette route sera associée à la méthode **vider** et devra utiliser le service **PanierService** pour vider complètement le panier.
 - L'action **vider** devra rediriger l'utilisateur vers la route **app_panier_index**
 - Modifiez le *template* **panier/index.html.twig** pour offrir un bouton  qui déclenchera la route **app_panier_vider**

🔗 Quelques indications utiles :

- Si vous trouvez pénible de devoir rajouter le paramètre `_locale` dans chaque URL d'une nouvelle route à l'intérieur d'une classe contrôleur, vous avez la possibilité de préciser, par une annotation **Route** au niveau de la classe, un préfixe qui se rajoutera automatiquement devant l'URL de chaque route définie dans la classe :

```
#[Route(path: '/_{locale}/panier', requirements: ['_locale'=> '%app.supported_locales%'])]
```



```
class PanierController extends AbstractController { ... }
```
- Dans une route, on peut contraindre un paramètre et vérifier qu'il respecte bien une expression régulière. Par exemple, pour vérifier qu'un paramètre de type **id** doit être composé uniquement de chiffres, on écrirait :

```
#[Route('/afficher/{id}', name: 'une_route', requirements: ['id' => '\d+'])]
```

3.3. Contrôleur Imbriqué dans la Barre de Navigation

On souhaite maintenant rendre dynamique la barre de navigation du site afin que le nombre de produits ajoutés au panier soit en permanence affiché. Pour cela, vous devrez :

- Développer, dans la classe contrôleur **PanierController**, un nouveau contrôleur :
`public function nombreProduits(PanierService $panier): Response { // à compléter... }`

Ce contrôleur devra renvoyer une réponse contenant uniquement le nombre de produits présents dans le panier.

Ce contrôleur sera un « contrôleur imbriqué » destiné uniquement à être appelé depuis un *template* et il ne sera donc associé à aucune route (voir **cours 03, pages 15-17**).

👉 **Rappel** : un contrôleur peut renvoyer simplement une réponse, sans faire appel à un *template*, en exécutant :
`return new Response("Hello World") ;`

- Modifier le *template* **navBar.html.twig** afin que la barre de navigation comporte un nouveau bouton permettant d'afficher le contenu du panier, et donc de déclencher la route **app_panier_index**.

Le texte de ce bouton devra afficher le nombre de produits dans le panier et ceci pourra être réalisé en appelant, depuis le *template*, le contrôleur imbriqué **PanierController::nombreProduits**

👉 **Rappel** : nous avons convenu de ne plus traduire les nouvelles pages, mais nous avons aussi convenu que le contenu de la barre de navigation devait, lui, être traduit... Il faudra donc faire le nécessaire (cf TP 02) pour pouvoir traduire le texte de ce nouveau bouton de navigation vers le panier !

4.0. Travail à réaliser – Vue d'ensemble

L'objectif de ce TP est de remplacer la source de données temporaire que nous utilisons jusqu'alors, c'est-à-dire le service BoutiqueService, par des entités persistantes de Doctrine : **Categorie** et **Produit**. L'accès à ces données se fera toujours *via* des services : les **Repository** que Doctrine fournit pour chaque type d'entité.

A la fin de ce TP, vous devrez supprimer le service BoutiqueService et votre application devra fonctionner comme auparavant, mais avec des données qui proviendront maintenant d'une BD (MariaDB).

4.1. Configurez votre projet pour utiliser Doctrine

Vous devez tout d'abord configurer votre projet Symfony pour lui indiquer quel serveur de BD vous allez utiliser, quelle sera la BASE sur ce serveur et avec quel LOGIN/PASSWORD vous vous y connecterez. Pour cela vous devez modifier, dans le fichier **.env** présent à la racine de votre projet, la variable **DATABASE_URL** de la façon suivante :

```
DATABASE_URL=mysql://LOGIN:PASSWORD@ellsworth.iut2.upmf-  
grenoble.fr:3306/BASE?serverVersion=10.11.9-MariaDB&charset=utf8mb4
```

Dans votre fichier .env, le texte ci-dessus doit tenir sur une seule ligne !!

Où vous remplacerez :

- **LOGIN** par **etu_login** où **login** est votre login UGA habituel (8 caractères)
- **PASSWORD** par **votre numéro étudiant Apogée** (8 chiffres)
- **BASE** par **symfony_login** où **login** est votre login UGA habituel (8 caractères)

Si vous ne connaissez pas votre numéro étudiant Apogée, vous pouvez le retrouver ici :

<https://scolarite-informatique.iut2.univ-grenoble-alpes.fr/app/ficheEtudiant.php>

Une fois ce paramétrage réalisé, Doctrine pourra utiliser la BD indiquée pour y faire persister les entités de votre application. Vous pourrez consulter le contenu de cette BD (MariaDB, un *fork* de MySQL) en utilisant l'interface Web PhpMyAdmin disponible sur le serveur **ellsworth**, interface à laquelle vous vous connecterez en utilisant les identifiants que vous avez saisis dans le fichier **.env** : <https://iut2-dg-scolarite.iut2.univ-grenoble-alpes.fr/info/phpmyadmin/>

Attention : Sur **phpmyadmin**, si vous collez votre mot de passe après l'avoir copié depuis le site **scolarite-informatique**, un espace sera rajouté au bout : il faudra penser à l'enlever. Sinon le plus simple est de taper votre numéro apogée « à la main ».

4.2. Créez vos entités Categorie et Produit

- Dans le terminal, à la racine de votre projet, créez une première entité **Categorie** en utilisant la commande :
php bin/console make:entity (voir cours 04, page 5 et page 20)
Cette entité devra comporter les attributs suivants (*attention, utilisez bien les même noms d'attributs que ceux utilisés précédemment dans BoutiqueService pour que le « refactoring » ne soit pas pénible ensuite*) :
 - **libelle** (type string 255)
 - **visuel** (type string 255)
 - **texte** (type text)
- De la même façon, créez une deuxième entité **Produit** avec les attributs suivants :
 - **libelle** (type string 255)
 - **visuel** (type string 255)
 - **texte** (type text)
 - **prix** (type decimal, précision 10, scale 2)
 - **categorie** (type relation ManyToOne vers Categorie, avec relation inverse)
- Allez regarder le code que Symfony a produit pour vous dans les répertoires **src/Entity** et **src/Repository**.
- Il faut maintenant enregistrer cette modification de votre modèle de données dans une migration et exécuter cette migration sur le serveur SQL pour les changements soient pris en compte. Pour cela taper les 2 commandes suivantes :

```
php bin/console make:migration  
php bin/console doctrine:migrations:migrate
```

4.3. Peuplez votre BD avec des « fixtures »

Auparavant, nous disposions dans `BoutiqueService` de données (4 catégories, 12 produits) qui étaient stockées « en dur » dans des tableaux. C'était bien utile pour tester notre code. Nous aimerions maintenant disposer de ces mêmes données, sous forme d'entités, pour pouvoir continuer à tester notre code. Symfony propose un mécanisme, appelé **Fixtures**, pour disposer, dans un projet, d'entités de test qui pourront être créées à la demande. Pour utiliser ce mécanisme, il faut installer un bundle Symfony par :

```
symfony composer require --dev orm-fixtures
```

Les entités « fixtures » doivent être créées programmatiquement dans une classe `AppFixtures.php` qu'il faudra placer dans le répertoire `src/DataFixtures`. On vous fournit ce répertoire et son contenu, sur Chamilo, fichier : `DataFixtures.zip`. Vous devez en décompresser le contenu et le placer dans le répertoire `src` de votre projet.

Regardez le code fourni dans `AppFixtures.php` : il vous montre un exemple de création d'entités avec Doctrine.

Pour exécuter ce code et « charger » dans la BD vos entités de test, utilisez la commande :

```
php bin/console doctrine:fixture:load
```

Vous pouvez alors aller consulter le contenu de vos tables dans votre BD et vérifier que les entités « fixtures » ont bien été sauvegardées en base.

4.4. Refactoring : se passer du service `BoutiqueService`

Nous sommes prêts maintenant à nous passer du service `BoutiqueService` et à utiliser à la place les `Repository` de Doctrine que nous avons peuplé avec nos données de test.

Vous devez donc maintenant reprendre votre code et, à chaque endroit où vous utilisiez `BoutiqueService`, vous devez maintenant utiliser à la place les services `CategorieRepository` et `ProduitRepository` (cf cours 04, pages 12-15)

Normalement, si vous avez respecté les consignes données en TP, le code concerné se trouve dans les classes :

- `BoutiqueController`
- `PanierController`
- `PanierService`

4.5. Définir une requête de recherche de produit dans `ProduitRepository`

Si vous avez mis en place dans la barre de navigation le champ « recherche de produit », vous avez dû utiliser, dans `BoutiqueController`, la méthode `findProduitByLibelleOrTexte` du service `BoutiqueService`.

Il n'y a pas d'équivalent à cette méthode parmi les méthodes qui sont fournies par défaut dans `ProduitRepository` et vous allez donc devoir la développer (voir cours 04, pages 16-18).

Dans le fichier `src/Repository/ProduitRepository.php`, développez une nouvelle méthode :

```
/**
 * @return Produit[] Returns an array of Produit objects
 */
public function findByLibelleOrTexte(string $recherche): array {
    // à compléter
}
```

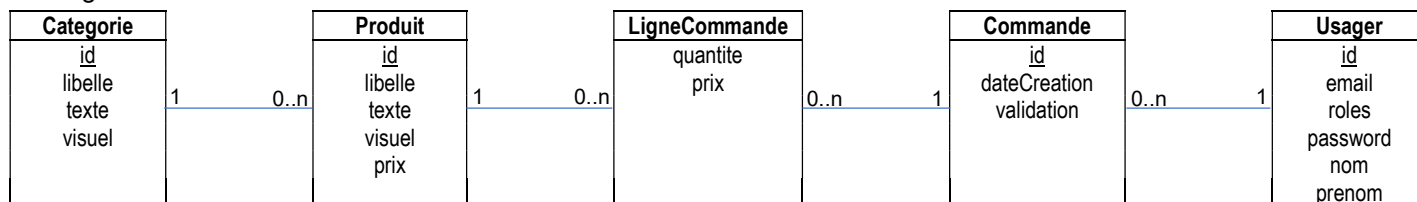
Indice : Il faut utiliser l'opérateur DQL/SQL « like » pour chercher tous les produits dont le libellé ou le texte contient la chaîne `$recherche`

4.6. Supprimer le service `BoutiqueService`

Lorsque tout fonctionne à nouveau comme avant, mais grâce à Doctrine, vous pourrez supprimer le fichier `src/Service/BoutiqueService.php` de votre projet, il ne nous sert plus à rien !

5.0. Travail à réaliser – Vue d'ensemble

Il faut maintenant donner la possibilité à un usager enregistré de transformer un panier en commande. Pour cela, il faut compléter notre « modèle » en y rajoutant les entités **Usager**, **Commande** et **LigneCommande**. Le diagramme de classe des entités devra être le suivant :



Pour l'instant, nous n'avons pas mis en place la « sécurité » de notre application, c'est-à-dire l'authentification de l'utilisateur *via* le composant « sécurité » de Symfony. En attendant cette étape, nous allons, dans un premier temps :

- Donner la possibilité de créer un nouvel **Usager**
- Mettre en place la validation du panier, validation qui consistera à :
 - Créer une commande pour l'utilisateur d'identifiant égal à 1 (*c'est temporaire, en attendant le TP6*)
 - Créer autant de lignes de commande qu'il y a d'articles dans le panier

Par la suite nous modifierons ce code lorsque nous aurons mis en place l'authentification pour que ce soit l'utilisateur authentifié qui passe commande. La modification sera très rapide à réaliser !

5.1. Entité Usager : Grimpons sur l'Echaffaudage

L'entité **Usager** est un peu particulière car c'est l'entité qui sera utilisée pour mettre en place l'authentification par la suite. Elle devra avoir des propriétés particulières. Pour cela :

- Taper la commande suivante pour installer un package nécessaire : `composer require doctrine/dbal:^3.8`
- Créer une entité nommée **Usager** par la commande : `php bin/console make:user Usager`
 - Préciser que c'est une entité persistante « Doctrine »
 - Préciser que la propriété **email** de cette entité sera celle utilisée plus tard pour l'authentification
 - Préciser que le mot de passe sera encrypté
- L'entité **Usager** a été créée avec les propriétés nécessaires à la « sécurité » de symfony (**email**, **password**, **roles**).
- On peut maintenant la compléter par la commande habituelle : `php bin/console make:entity Usager`
 - Rajouter les propriétés **nom** et **prenom**
- Répercuter ces changements sur la BD par les commandes habituelles :
 - `php bin/console doctrine:migration:diff`
 - `php bin/console doctrine:migration:migrate`

Pour pouvoir rapidement disposer d'un formulaire permettant de créer un nouvel utilisateur, on va se servir de la fonctionnalité CRUD de symfony :

- Créer l'interface CRUD pour l'entité **Usager** par la commande : `php bin/console make:crud`

Il ne reste plus qu'à adapter le code ainsi créé aux besoins de notre maquette :

- Dans le contrôleur `src/Controller/UsagerController.php` :
 - Ne conserver que les méthodes :
 - **index** (affichage de la page d'accueil de l'utilisateur)
 - **new** (formulaire d'inscription d'un nouvel utilisateur)
 - Modifier les annotations `#route` pour y ajouter la locale (*cf* TP5 précédents)
 - Modifier la méthode **index** pour qu'elle transmette à son *template* l'entité **Usager** dont l'identifiant est égal à 1 (*c'est temporaire, en attendant le TP6*)
 - Modifier la méthode **new** pour qu'elle encrypte le mot de passe, à l'aide du service **UserPasswordEncoderInterface**, avant de faire persister le nouvel **Usager** :

```

if ($form->isSubmitted() && $form->isValid()) {
    // Encoder le mot de passe qui est en clair pour l'instant
    $hashedPassword = $passwordHasher->hashPassword($usager, $usager->getPassword());
    $usager->setPassword($hashedPassword);
    // Définir le rôle de l'utilisateur qui va être créé
    $usager->setRoles(["ROLE_CLIENT"]);
    //...
}
  
```

- Modifier la méthode **new** pour qu'à la fin elle redirige vers la route `app_usager_index`

- Dans le répertoire `templates/usager` :
 - Ne conserver que les templates `index.html.twig`, `new.html.twig` et `_form.html.twig`
 - Modifier le template `index.html.twig` pour qu'il affiche le nom et prénom de l'Usager qui lui sera transmis (s'il existe).
 - Supprimer les liens de navigation vers les pages du CRUD dans les templates `index.html.twig` et `new.html.twig`
- Dans le formulaire `src/Form/UsagerType.php` :
 - Modifier la méthode `buildForm` pour que seuls les champs email, password, nom et prénom figurent dans le formulaire qui sera soumis à l'utilisateur

Modifier enfin la barre de navigation pour proposer un menu déroulant Usager avec deux entrées « inscription » et « accueil » qui pointeront respectivement vers les routes `app_usager_new` et `app_usager_index`

5.2. Entités Commande et LigneCommande

- Créer ces entités à l'aide de la commande habituelle `php bin/console make:entity`
- L'entité **Commande** possède une relation **ManyToOne** vers l'entité **Usager**
- L'entité **LigneCommande** possède :
 - Une relation **ManyToOne** vers l'entité **Commande**
 - Une relation **ManyToOne** vers l'entité **Produit**
- Répercuter ces changements sur la BD par les commandes habituelles :
 - `php bin/console doctrine:migration:diff`
 - `php bin/console doctrine:migration:migrate`

5.3. Transformer un Panier en Commande

- Dans votre service `src/Service/PanierService.php` :
 - Ajouter une nouvelle méthode `panierToCommande` qui reçoit en paramètre une entité de type **Usager** et qui crée, pour cet usager, une commande (et ses lignes de commande) à partir du contenu du panier (s'il n'est pas vide).
 - Le contenu du panier devra être supprimé à l'issue de ce traitement.
 - Cette méthode renverra en résultat l'entité **Commande** qui aura été créée

```
public function panierToCommande(Usager $usager) : ?Commande {
    // à compléter
}
```

- Créer une route `app_panier_commander` et une méthode `commander` dans le contrôleur **PanierController** :
 - Cette action devra bien sûr utiliser la méthode `panierToCommande` créée précédemment
 - Elle utilisera (*temporairement*) l'usager d'identifiant égal à 1 comme propriétaire de la commande
 - Elle se terminera par l'affichage d'un *template* `commande.html.twig` qui indiquera à l'utilisateur son prénom, son nom, le numéro et la date de la commande qu'il vient de passer

6.0. Travail à réaliser – Vue d'ensemble

Terminez les TP précédents avant de réaliser cette étape.

L'objectif de cette séance est de mettre en place la sécurité sur votre application en suivant le processus présenté dans le cours 06... Ce sera rapide !

6.1. Commande `make:security` (cours 06 page 12)

- Exécutez la commande `php bin/console make:security:form-login`
- Attention à bien indiquer l'entité utilisée pour l'authentification : `App\Entity\User` (c'est parfois demandé par la commande... et parfois non !?)
- Cette commande crée (ou modifie) 3 fichiers qu'il va falloir adapter :
 - `src/Controller/SecurityController.php`
 - `templates/security/login.html.twig`
 - `config/packages/security.yaml`

6.2. Adapter `SecurityController` (cours 06 page 13)

- Si vous aviez auparavant mis en place la traduction sur votre site, modifiez les routes des deux contrôleurs `login` et `logout` présents dans la classe contrôleur `SecurityController` afin que ces routes gèrent l'internationalisation (elles devront avoir un paramètre `_locale`)

6.3. Adapter `login.html.twig` (cours 06 page 14)

- Modifiez le `template login.html.twig` pour que le formulaire d'authentification soit cohérent avec votre site (textes en français, style « bootstrap » pour les champs)

6.4. Adapter le formulaire `login.html.twig` (cours 06 page 15)

- Modifiez le `template login.html.twig` pour que le formulaire d'authentification soit cohérent avec votre site (textes en français, style « bootstrap » pour les champs)

6.5. Adapter la configuration `security.yaml` (cours 06 page 15)

- Définissez la hiérarchie des rôles (`ROLE_ADMIN` et `ROLE_CLIENT`, même si nous n'utiliserons pas `ROLE_ADMIN`)
- Dans la section `logout` du firewall `main`, définissez la route vers laquelle l'utilisateur sera redirigé après une déconnexion
- Configurez soigneusement la section `access_control` afin que les routes suivantes (et seulement elles !) nécessitent une authentification avec un rôle `ROLE_CLIENT` :
 - `app_user_index` (contrôleur qui affiche la page d'accueil d'un usager authentifié)
 - `app_panier_commander` (contrôleur qui transforme un panier en commande)

6.6. Refactoring : utiliser l'utilisateur authentifié lors d'une commande

- Modifiez le contrôleur `PanierController::commander` afin que ce soit maintenant l'utilisateur authentifié qui se voit attribuer la commande qui est passée (*et non plus par défaut l'utilisateur numéro 1 comme c'était demandé auparavant*)
- Pour accéder à l'utilisateur connecté dans un contrôleur : voir cours 06 page 17

6.7. Adapter l'interface

- Ajoutez un choix « authentication » dans le menu Usager de votre navBar
- Ajoutez un choix « déconnexion » dans le menu Usager de votre navBar
- Modifiez la barre de navigation et vos *templates* pour qu'ils s'adaptent selon qu'un utilisateur est authentifié ou pas :
 - Le bouton « passer commande » du panier ne doit être visible que si l'utilisateur est authentifié
 - Les choix proposés dans le menu Usager de la navBar doivent être activés ou désactivés en fonction de l'authentification
- Pour accéder à l'utilisateur connecté dans un template Twig : voir cours 06 page 17

6.8. Facultatif

S'il vous reste du temps, développez :

- Une page qui permet à l'utilisateur connecté de consulter la liste de ses commandes (accessible depuis le menu Usager de la navBar)
- Une page qui permet à l'utilisateur connecté de consulter le détail d'une de ses commandes (accessible depuis la page ci-dessus)
- Pensez à bien sécuriser ces pages qui ne doivent être accessibles que si l'utilisateur authentifié possède le `ROLE_CLIENT`

7.1. SideBar « Top Ventes »

Votre site devrait afficher, sur chaque page, une liste des n produits les plus vendus ($n=3$ par exemple).

Pour cela vous devrez :

- Programmer dans un Repository (lequel ? à vous de trouver !) la requête qui permet de récupérer la liste des n produits les plus vendus. Vous aurez besoin d'utiliser SUM et GROUP BY dans votre requête : <https://www.doctrine-project.org/projects/doctrine-orm/en/2.11/cookbook/aggregate-fields.html>
- Développer un contrôleur imbriqué qui devra utiliser cette requête pour alimenter le petit *template* qui permettra d'afficher cette liste dans une *SideBar*
- Utiliser ce contrôleur imbriqué dans votre *layout* : `base.html.twig`

7.2. Captcha

- Pour vérifier que, lorsqu'un client s'inscrit sur notre site, il s'agit bien d'un humain, il faudrait mettre en place un CAPTCHA dans le formulaire d'inscription.
- En voici un facilement intégrable à Symfony : <https://github.com/Gregwar/CaptchaBundle>
- Il suffit de... RTFM 😊

7.3. Gestion des Devises

Votre site devrait permettre à l'utilisateur d'afficher les prix dans la devise de son choix. Il faut donc être capable de convertir les prix en euros traités par votre application en fonction des taux de change du jour, fournis par la Banque Centrale Européenne (BCE).

Pour cela vous devrez :

- Disposer d'une variable de session qui contient la devise courante actuellement utilisée pour l'affichage des prix. Une devise est représentée par une chaîne de 3 caractères, par exemple « EUR » ou « USD »
- Proposer dans votre barre de navigation un menu déroulant qui permet de choisir la devise courante parmi une liste prédéfinie de devises (par exemple EUR, USD, GBP)
- Installer l'extension twig IntlExtension : `composer req twig/intl-extra`
Cette extension vous permettra de disposer du filtre `format_currency` qui permet d'afficher une somme dans une devise donnée : https://twig.symfony.com/doc/3.x/filters/format_currency.html
Cette extension doit être déclarée comme service dans le fichier `config/services.yaml` :

```
services:
    # ...
    twig.extension.intl:
        class: Twig\Extra\Intl\IntlExtension
        tags:
            - { name: twig.extension }
```
- Développer un service `DeviseService`.
Ce service devra fournir une unique méthode permettant de convertir une somme en euros en une somme dans la devise courante définie en session.
Pour cela, votre service devra utiliser un tableau de conversion censé être lui aussi disponible en session. S'il ne l'est pas, il faudra aller chercher (`file_get_contents`) les taux de change via le Web Service de la BCE et les mettre en session.
Après [inscription gratuite](#) pour obtenir une clé XXXX, vous pourrez utiliser l'URL suivante :
`http://api.exchangeratesapi.io/v1/latest?access_key=XXXX&format=1`
Elle vous permettra de récupérer un objet JSON très simple à utiliser (`json_decode`), contenant les taux de conversion.
- Développer une extension Twig afin de disposer, dans vos *templates*, d'un nouveau filtre `currency_convert` capable de convertir une somme en euros en une somme dans la devise courante.
Ce filtre devra bien sûr utiliser le service `DeviseService` écrit auparavant.
Documentation pour définir un nouveau filtre : https://symfony.com/doc/5.4/templating/twig_extension.html
- Lorsque tout sera en place, vous n'aurez plus qu'à modifier vos *templates* pour appliquer, à chaque prix affiché, les deux filtres nécessaires : d'abord votre filtre `currency_convert` pour convertir la somme, puis le filtre `format_currency` pour afficher correctement la somme convertie.