Organisation des répertoires rencontrées pendant les TP situées à la racine du projet Symfony (IDE PhpStorm)

Installation des dépendances: composer install

- public
- src/Controller
- src/Entity
- src/Repository
- src/Service
- templates (TP1)
- translations

Notes de TP - Développement d'un site de e-commerce avec le framework PHP Symfony et le moteur de templates Twig

TP1

Construction des premières pages avec le moteur de template twig

- Un template Twig est un fichier contenant des balises Twig qui seront interprétées par le moteur de rendu.
- Le template peut être de n'importe quel format texte : HTML, XML, PDF.
- Les balises Twig peuvent être insérées n'importe où dans le texte.

Twig définit trois sortes de balises :

- `{{ ... }}` : affiche une variable ou le résultat d'une expression.
- `{% ... %}` : contrôle la logique du template, utilisé pour exécuter des boucles for et des conditions if, entre autres.
- `{# ... #}` : commentaires.

Le titre, équivalent à la balise html

```
{% block title %}
{% endblock %}
```

```
<title>
</title>
```

Le corps de la page, équivalent à la balise html

```
{% block body %} {% endblock %}
```

```
<body>
</body>
```

Twig - Héritage

Un template peut être « décoré » par un autre mécanisme d'héritage. L'héritage de template permet de bâtir un template « layout » de base qui contient tous les éléments communs de votre site et de définir des « blocks » que les templates enfants pourront surcharger.
Le tag `{% block %}` définit des blocs que les templates enfants vont pouvoir redéfinir.
EXEMPLE AVEC LA PAGE CONTACT FAITE PENDANT LE TP1

```
{# templates/default/contact.html.twig #}

{% extends 'base.html.twig' %}

{# TODO #}
{% block title %}Contact{% endblock %}

{% block body %}
    [...]
{% endblock %}
```

Twig - Inclusion

Pour partager un morceau de vue (un fragment) entre plusieurs templates distincts, on peut créer un nouveau template qui sera inclus dans les autres.

Twig - Liens (plus de détails dans la suite)

Twig - Assets

-> Pour utiliser des ressources (images, css, ...) dans un templateTwig de façon « portable », il faut utiliser la fonction asset.
->Grâce à cette fonction, on peut déplacer le répertoire de votre application sans devoir changer le code des templates.
->Les ressources (images, css, ...) doivent être placées dans le répertoire public de votre projet.
->Le chemin passé en paramètre à asset est un chemin relatif par rapport au répertoire public.
EXEMPLE AVEC LES LIENS ET LES ASSETS DU TP1

```
{# templates/base.html.twig #}
<!DOCTYPE html\>
<html lang="fr">
```

```
<head>
[...]
    <link rel="icon" href="{{ asset('images/favicon.ico') }}">
    <title>
        {% block title %} TODO : définir contenu block title {% endblock %}
    </title>
    <!-- CSS : Bootstrap, Fontawsome, projet (styles communs à tout le
projet) -->
    <link href="{{ asset('css/bootstrap-4/bootstrap.min.css') }}"
rel="stylesheet">
    <link href="{{ asset('css/fontawsome-5.all.min.css') }}"
rel="stylesheet">
    <link href="{{ asset('css/projet.css') }}" rel="stylesheet">
    <!-- début du block css spécifique -->
    {% block css %}{%endblock %}
    <!-- fin du block css -->
</head>
<body>
    <!-- début de la navbar -->
    {% include "navbar.html.twig" %}
    <!-- fin de la navbar -->
    <main role="main" class="container-fluid">
        <div class="row">
            <div class="col-md-10">
                <div
                    class="main-div">
                    <!-- début du block body -->
                    {%  block body %}
                        <h1>
                            <i>TODO : définir contenu block body</i>
                        </h1>
                    {%  endblock %}
                    <!-- fin du block body -->
                </div>
            </div>
        </div>
    </main>
    <!-- JS : JQuery, Bootstrap -->
    <script src="{{ asset('js/jquery-3/jquery.slim.min.js') }}"></script>
    <script src="{{ asset('js/bootstrap-4/bootstrap.bundle.min.js') }}">
</script>
    <script>
        $(function () {
            // Activation tooltip bootstrap sur la page
            $('[data-toggle="tooltip"]').tooltip();
            // Script du bouton de recherche
            // TODO
        });
    </script>
    <!-- debut du block js spécifique -->
    {% block js %}{%  endblock %}
    <!-- fin du block js spécifique -->
</body>
</html>
```

Coder un contrôleur (BoutiqueController.php)

BoutiqueController créée avec la commande `bin/console make:controller`

Exemple de la méthode index dans la classe BoutiqueController

```
#[Route(
    path: '/boutique',
    name: 'app_boutique'
)]
public function index(BoutiqueService \$boutiqueService): Response {
    // Utilisation du service BoutiqueService.php pour trouver toutes les
catégories existantes
    $categories = $boutiqueService->findAllCategories();
    // render est une fonction de Symfony qui renvoie un objet de type
Response qui est le fichier twig
    return $this->render('boutique/index.html.twig', [
        "categories" => $categories,
    ]);
}
```

Remarques: -> Chaque méthode d'un contrôleur a une route qui lui est associée.

-> Une route possède au minimum un attribut path (chemin de la route) et un attribut name (nom associé à la route).

-> L'attribut name est utilisé pour référencer les liens dans les templates twig.

-> La fonction a deux paramètres : le premier est la vue (Template twig) qui doit être renvoyée et le deuxième paramètre (optionnel) est une variable utilisable dans la vue (ici boutique/index.html.twig) de nom catégories qui correspond la variable php $categories définie plus haut dans la fonction.

Voici des exemples tirès de boutique/index.html.twig qui illustrent les deux dernières remarques.

```
<a class="btn-back text-secondary" href="{{ path('app_default_index') }}">
    <i class="fas fa-x fa-arrow-circle-left"></i>
</a>

{% if categories | length < 10 %}
    0{{ categories | length }}
{% else %}
    {{ categories | length }}
{% endif %}
{.badge .badge-pill .badge-info}

<div class="card-body row">
{% for categorie in categories %}
    <div class="card clickable" onclick=" location = '{{
path("app_boutique_rayon{idCategorie:categorie.id}) }}'">
        <img src="{{ categorie.visuel }}" class="card-img-top" alt="{{
categorie.libelle }}">
```

```
            <div class="card-body">
                <h4 class="card-title">{% trans
%}boutique.index.categorie_libelle{% endtrans %}{{ categorie.libelle }}</h4>
                <p class="card-text">
                    <i>{% trans %}boutique.index.categorie_texte{% endtrans
%}{{ categorie.texte }}</i>
                </p>
            </div>
        </div>
    </div>
{% endfor %}
```

TP2

Objectif: 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 comportera un petit menu
déroulant permettant de choisir la langue affichée sur le site
Partie 1 - Traduction des pages dans différentes langues

1. Définition de la locale par défaut et de la locale de repli dans le
   fichier config/packages/translation.yaml

config/packages/translation.yaml
framework: default_locale: 'fr' translator: default_path:
'%kernel.project_dir%/translations' fallbacks: ['fr'] providers:

2. Définition des langues supportées par l'application dans le fichier
   config/services.yaml

This file is the entry point to configure your own services.
Files in the packages/ subdirectory configure your dependencies.
Put parameters here that don't need to change on each machine where the app is deployed
https://symfony.com/doc/current/best_practices.html#use-parameters-for-application-configuration
parameters: app.supported_locales: 'fr|en|ja' //MODIFIER CETTE LIGNE
services: # default configuration for services in *this* file
_defaults: autowire: true # Automatically injects dependencies in your
services. autoconfigure: true # Automatically registers your services
as commands, event subscribers, etc.

```
# makes classes in src/ available to be used as services
# this creates a service per class whose id is the fully-qualified class
name
App\:
    resource: '../src/'
    exclude:
        - '../src/DependencyInjection/'
        - '../src/Entity/'
        - '../src/Kernel.php'

# add more service definitions when explicit configuration is needed
# please note that last definitions always *replace* previous ones
```

3. Modification des routes dans les contrôleurs pour prendre en compte la langue (la locale est 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.
Exemple de la méthode index() dans la classe DefaultController
#[Route(path: '/{_locale}/', name: 'app_default_index', requirements: ['_locale' => '%app.supported_locales%'], defaults: ['_locale' => '%default_locale%'])] public function index(): Response { return $this->render('default/index.html.twig'); }

4. Modifier les templates en entourant les parties à traduire des balises {% trans %} texte à traduire {% endtrans %} Les éléments sont annotés par un identifiant de la forme repertoire_du_template.nom_du_template.identifiant_element_a_traduire

Exemple tiré de la navbar dans le fichier templates/navbar.html.twig
[...]

```
<li class="nav-item">
```

<a class="nav-link" href="{# TODO #}{{ path("app_boutique") }} `<i class="fas fa-store">` {=html} `</i>` {=html} {% trans %}templates.navbar.boutique{% endtrans %} `</a>` {=html}

```
</li>
```

[...]

5. On génére les catalogues Symfony avec la commande php bin/console translation:extract --force LL où LL est un identifiant de langue défini précédemment dans config/services.yaml. Ces catologues vont servir à lier les identifiants insérées dans les templates twig via la balise trans au contenu de ces catalogues selon la locale présente dans l'URL.

Commandes tapées durant le TP php bin/console translation:extract --force fr php bin/console translation:extract --force en php bin/console translation:extract --force ja

6. L'étape précédente produit un catalogue (fichier XLIFF) nommé translations/messages+intl-icu.LL.xlf. Dans chacun de ces catalogues, remplissez les balises `<target>` {=html}, associées à chaque balise `<source>` {=html}, afin d'indiquer par quoi sera remplacé l'identifiant défini dans la source lorsqu'il sera affiché avec la locale LL

Fichier messages+intl-icu.fr.xlf
`<trans-unit id="0z7fGq8" resname="templates.navbar.boutique">` {=html}
`<source>` {=html}templates.navbar.boutique `</source>` {=html}
`<target>` {=html}Boutique `</target>` {=html} `</trans-unit>` {=html}
Fichier messages+intl-icu.en.xlf
`<trans-unit id="0z7fGq8" resname="templates.navbar.boutique">` {=html}
`<source>` {=html}templates.navbar.boutique `</source>` {=html}
`<target>` {=html}Store `</target>` {=html} `</trans-unit>` {=html}
Fichier messages+intl-icu.ja.xlf
`<trans-unit id="0z7fGq8" resname="templates.navbar.boutique">` {=html}
`<source>` {=html}templates.navbar.boutique `</source>` {=html}
`<target>` {=html}お店 `</target>` {=html} `</trans-unit>` {=html}
Partie 2: Menu des langues dans la barre de navigation
-Il faut avoir accès dans le template à la liste des locales supportées par l'application. Cette liste est injectée dans Twig en complétant le fichier config/packages/twig.yml : #config/packages/twig.yml twig: globals: supported_locales: '%app.supported_locales%'

-On dispose alors, dans les templates Twig, d'une variable supported_locales qui est une chaîne de caractères ('fr|en|ja') contenant les locales supportées, séparées par le caractère '|'. Pour parcourir ces locales dans une itération, on utilise 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 %} {% set localeCourante = app.request.attributes.get('_locale') %} {% set routeActuelle = app.request.attributes.get('_route') %} {% set routeParams = app.request.attributes.get('_route_params') %} [...]
`<!-- Début menu des langues -->`{=html}

```
<ul class="navbar-nav my-0">
```

```
<li class="nav-item dropdown">
```

[`<img src="" alt="{{ localeCourante }}">`{=html}]{.nav-link .dropdown-toggle data-toggle="dropdown" aria-haspopup="true" aria-expanded="false"}

```
<div class="dropdown-menu" aria-labelledby="dropdown01" style="min-width:
5rem;">
```

```
            {% for uneLocale in supported_locales | split('|') %}
                {% set routeParams = routeParams | merge({'_locale':
 uneLocale }) %}
                {% if uneLocale != localeCourante %}
                    <a class="dropdown-item" href="{{ path(routeActuelle,
 routeParams) }}">
                        <img src="" alt="{{ uneLocale }}">
                    </a>
                {% endif %}
            {% endfor %}
            </div>
        </li>
    </ul>
```

```
<!-- Fin Menu des langues -->
```

Partie 3: Champ de recherche dans la barre de navigation
// TODO let searchField= document.querySelector("#searchString"); let searchBtn = document.querySelector("#searchButton");
searchBtn.addEventListener("click", (event) => {
event.preventDefault(); let encodeString = encodeURIComponent(searchField.value.trim()); if(encodeString){
window.location.href = "{{ path("app_boutique_chercher") }}" + "/" + encodeString; }else{ window.location.href = "{{ path("app_boutique_chercher") }}"; }
});
#[Route(path: '/chercher/{recherche}', name: 'app_boutique_chercher', requirements: ['recherche' => '.+'], // regexp pour avoir tous les car, / compris defaults: ['recherche' => ''])] public function chercher(ProduitRepository $produitRepository, string $recherche): Response { $searchString = urldecode($recherche); $produitsTrouves = $produitRepository->findProduitsByLibelleOrTexte($searchString); $nombreProduit = count($produitsTrouves);

```
        return $this->render('boutique/chercher.html.twig', [
            'searchString' => $searchString,
            'produits' => $produitsTrouves,
            'nombreProduit' => $nombreProduit,
        ]);
    }
```

TP3 - Contrôleur, service, session

Développement de la classe PanierService (Fichier src/Service/PanierService.php)
Méthode sur les sessions en Symfony private $session; // Le service session private $boutique; // Le service boutique private $panier; // Tableau associatif, la clé est un idProduit, la valeur associée est une quantité // donc $this->panier[$idProduit] = quantité du produit dont l'id = $idProduit const PANIER_SESSION = 'panier'; // Le nom de la variable de session pour faire 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, init. à vide sinon $this->panier = $this->session->get(self::PANIER_SESSION, array()); }
la méthode get() sur un objet session permet de récupérer un objet stocké sur la session qu'on a besoin de faire persister. Cette méthode prend 2 paramètres (le 2ème est optionnel). Le premier paramètre est le nom (identifiant) de l'objet en session et le second est sa valeur par défaut dans le cas où l'objet n'existerait pas.
Il ne faut pas oublier de mettre à jour l'objet qui est stocké en session (ici, c'est le panier) à chaque manipulation de celui-ci.
private function mettreAJourPanier() : void { $this->session->set(self::PANIER_SESSION, $this->panier); }
On appelle cette méthode dans chaque méthode qui agit sur le panier. Par exemple, dans les méthodes qui ajoute et enlève un produit au panier. // Ajouter au panier le produit $idProduit en quantite $quantite public function ajouterProduit(int $idProduit, int $quantite = 1) : void { /* A COMPLETER */ if(isset($this->panier[$idProduit])){ $this->panier[$idProduit] += $quantite; }else{ $this->panier[$idProduit] = $quantite; } $this->mettreAJourPanier(); }
// Enlever du panier le produit $idProduit en quantite $quantite public function enleverProduit(int $idProduit, int $quantite = 1) : void { if(isset($this->panier[$idProduit]) && $this->panier[$idProduit] > $quantite){ $this->panier[$idProduit] -= $quantite; }else{ unset($this->panier[$idProduit]); } $this->mettreAJourPanier(); }
Pour se connecter à la bd de l'iut avec machine perso
Connexion ssh
mais le + évident, c'est d'avoir un WAMP sur Windows ou d'avoir déjà d'installer PostgreSQL ou SQLite et modifier le .env. ATTENTION ! Ne pas importer le .env sur l'espace de l'iut.

TP4 - Mise en place de données persistantes dans une BD avec Doctrine

0. Configuration -> Il faut configurer le projet Symfony pour lui indiquer quel serveur de BD utiliser, quelle sera la base sur ce serveur et avec quel LOGIN/PASSWORD s'y connecter. Il faut modifier la variable DATABASE_URL présent dans le fichier .env du projet comme par exemple:

DATABASE_URL=mysql://etu_drouichi:12111023@ellsworth.iut2.upmf-grenoble.fr:3306/symfony_drouichi?serverVersion=10.11.9-MariaDB&charset=utf8mb4
ou encore
DATABASE_URL=postgresql://ilyes@127.0.0.1:5432/tp04_symfonydoctrine

Notions à connaitre: -> Entity Symfony (Data Object) : -> Repository Symfony (Data Access Object = DAO) :

1) Créer une classe Entité

Entité Catégorie TP4
php bin/console make:entity Class name of the entity to create or update: > Categorie
New property name (press `<return>` {=html} to stop adding fields): > libelle Field type (enter ? to see all types) [string]: > string
Field length [255]: > 255 Can this field be null in the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): > visuel Field type (enter ? to see all types) [string]: > string Field length [255]: > 255 Can this field be null in the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): > texte Field type (enter ? to see all types) [string]: > text Can this field be null in the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): >
Entité Produit TP4
php bin/console make:entity Class name of the entity to create or update: > Produit
New property name (press `<return>` {=html} to stop adding fields): > libelle Field type (enter ? to see all types) [string]: > string
Field length [255]: > 255 Can this field be null in the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): > visuel Field type (enter ? to see all types) [string]: > string Field length [255]: > 255 Can this field be null in the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): > texte Field type (enter ? to see all types) [string]: > text Can this field be null in the database (nullable) (yes/no) [no]: > no

New property name (press `<return>` {=html} to stop adding fields): >
prix Field type (enter ? to see all types) [string]: > decimal
Precision (total digits stored: e.g. 10) [10]: > 10 Scale (number of
decimal digits to store: e.g. 2) [0]: > 2 Can this field be null in
the database (nullable) (yes/no) [no]: > no
New property name (press `<return>` {=html} to stop adding fields): >
categorie Field type (enter ? to see all types) [string]: > relation
What class should this entity be related to?: > Categorie
hat type of r elationship is this?:

Type Description
ManyToOne Each Produit relates to one Categorie

ManyToOne
Is the Produit.categorie property allowed to be null (nullable)?
(yes/no) [yes]: > no
Do you want to add a new property to Categorie so that you can
access/update Produit objects from it -
e.g. $categorie->getProduits()? (yes/no) [yes]: > yes
A new property will also be added to Categorie: "produits" (an
ArrayCollection of Produit objects).
New property name (press `<return>` {=html} to stop adding fields): >
 Les objets de la classe PHP Product vont permettre de
représenter/manipuler des n-uplets d'une table product créée dans la BD.
permet de créer le code SQL pour mettre en place une nouvelle version de
vos entités (à exécuter à chaque modification de la structure des
entités) Chaque propriété (attribut) de la classe PHP correspond à une
colonne de la table SQL permet d'exécuter le code SQL pour mettre à jour
la BD (à exécuter à chaque modification de la structure des entités)
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
Point sur les relations
********** Type de relation
Lecture simplifiée Exemple concret Côté A (la classe actuelle) Côté B
(classe liée) ManyToOne Plusieurs A pour un B Plusieurs Livres
appartiennent à une Bibliothèque Livre → bibliotheque Bibliotheque →

livres (OneToMany) OneToMany Un A possède plusieurs B Une Bibliothèque contient plusieurs Livres Bibliotheque → livres Livre → bibliotheque (ManyToOne) OneToOne Un A a exactement un B (et inversement) Une Personne a un seul Passeport, et un Passeport appartient à une seule Personne Personne → passeport Passeport → propriétaire ManyToMany Plusieurs A peuvent avoir plusieurs B Un Étudiant suit plusieurs Cours et chaque Cours a plusieurs Étudiants Etudiant → cours Cours → étudiants Livre ----- (ManyToOne) -----> Bibliotheque Bibliotheque ----- (OneToMany) -----> Livres (Collection) Personne <-----> Passeport Etudiant <------> Cours

2) Peuplement de la BD avec le mécanisme de fixtures en Symfony

Exemple de code d'une classe Fixture src/DataFixtures/AppFixtures.php Pour exécuter ce code et « charger » dans la BD vos entités de test, utilisez la commande : php bin/console doctrine:fixture:load Pour vérifier, on peut aller consulter le contenu des tables dans la BD et vérifier que les entités « fixtures » ont bien été sauvegardées en base.

3) Utilisation des repository dans les contrôleurs PHP

Pour interagir avec les données de la BD qu'on a mise en place et qu'on vient tout juste de remplir. Il faut utiliser les repository qui contiennent les méthodes permettant de manipuler les données. Symfony fournit une palette de méthodes classiques par défaut sur les repository. Si cela ne suffit pour un traitement spécifique, il faut soi-même définir une requête comme vu en 4) juste après #[Route(path: '/{_locale}/boutique', requirements: ['_locale' => '%app.supported_locales%'])] final class BoutiqueController extends AbstractController { #[Route(path: '/', name: 'app_boutique')] public function index(CategorieRepository $categorieRepository): Response { $categories = $categorieRepository->findAll(); return $this->render('boutique/index.html.twig', ["categories" => $categories,]); }

```
#[Route(
    path: '/rayon/{idCategorie}',
    name: 'app_boutique_rayon',
    requirements: ['idCategorie' => '\d+']
)]
public function rayon(CategorieRepository $categorieRepository,
```

```
    ProduitRepository $produitRepository, int $idCategorie): Response
    {
        $categorie = $categorieRepository->find($idCategorie);
        $produits = $produitRepository->findBy(['categorie' => $categorie]);
        return $this->render('boutique/rayon.html.twig', [
            "categorie" => $categorie,
            "produits" => $produits
        ]);
    }

    #[Route(
        path: '/chercher/{recherche}',
        name: 'app_boutique_chercher',
        requirements: ['recherche' => '.+'], // regexp pour avoir tous les car,
    / compris
        defaults: ['recherche' => ''])
    ]
    public function chercher(ProduitRepository $produitRepository, string
    $recherche): Response
    {
        $searchString = urldecode($recherche);
        $produitsTrouves = $produitRepository-
    >findProduitsByLibelleOrTexte($searchString);
        $nombreProduit = count($produitsTrouves);

        return $this->render('boutique/chercher.html.twig', [
            'searchString' => $searchString,
            'produits' => $produitsTrouves,
            'nombreProduit' => $nombreProduit,
        ]);
    }
```

4) Intégrer et écrire une requête personnalisée dans un repository

Avec QueryBuilder:

/** * Recherche les produits contenant un mot-clé dans le libellé ou

le texte */ public function findProduitsByLibelleOrTexte(string

$recherche): array {

return

$this->createQueryBuilder('p') ->where('p.libelle LIKE :search') ->orWhere('p.texte LIKE :search') -

>setParameter('search', '%'.$recherche.'%')

->getQuery() ->getResult(); }

TP5 - Formulaires, validation, CRUD

php bin/console make:crud

Formulaire (de login) d'un usager (utilisateur) du site de E-commerce

src/Form/UsagerType.php autogénéré (avec quelques modifications) `<?php

namespace App\Form;

use App\Entity\Usager;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\OptionsResolver\OptionsResolver;

class UsagerType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options): void

{

$builder

->`{=html}add('email') ->add('password') ->add('nom')

->add('prenom'); }

```
 public function configureOptions(OptionsResolver $resolver): void
 {
     $resolver->setDefaults([
         'data_class' => Usager::class,
     ]);
 }
```

}

Template/usager/_form.html.twig (complétement autogénéré) {{

form_start(form) }} {{ form_widget(form) }}

`<button class="btn">`{=html}{{ button_label|default('Save')

}} `</button>`{=html} {{ form_end(form) }}

```
<?php

namespace App\Controller;

use App\Entity\Usager;
use App\Form\UsagerType;
use App\Repository\CommandeRepository;
use App\Repository\UsagerRepository;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
use Symfony\Component\Routing\Attribute\Route;

#[Route(
```

```
    path: '/{_locale}/usager',
    requirements: ['_locale' =>
```

'%app.supported_locales%'])] final class UsagerController extends AbstractController { #[Route(path:'/', name: 'app_usager_index', methods: ['GET'])] public function index(UsagerRepository $usagerRepository): Response { $user = $this->getUser(); $usager = $usagerRepository->find($user->getId());

```
    return $this->render('usager/index.html.twig', [
        "email" => $usager->getEmail(),
        "prenom" => $usager->getPrenom(),
        "nom" =>  $usager->getNom(),
        "nbCommandes" => count($usager->getCommandes()),
    ]);
}

#[Route(
    path: '/new',
    name: 'app_usager_new',
    methods: ['GET', 'POST'])]
public function new(Request $request, EntityManagerInterface
$entityManager, UserPasswordHasherInterface $passwordHasher): Response
{
    $usager = new Usager();
    $form = $this->createForm(UsagerType::class, $usager);
    $form->handleRequest($request);

    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'usager qui va être créé
        $usager->setRoles(["ROLE_CLIENT"]);
        //Persister et enregistrer en base
        $entityManager->persist($usager);
        $entityManager->flush();

        return $this->redirectToRoute('app_usager_index',
            ['usager' => $usager, 'form' => $form,],
Response::HTTP_SEE_OTHER);
    }

    return $this->render('usager/new.html.twig', [
        'usager' => $usager,
        'form' => $form,
    ]);
}
```

[...] }

Entité Commande

php bin/console make:entity Class name of the entity to create or update: > Commande

New property name (press `<return>` {=html} to stop adding fields): >

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

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

New property name (press `<return>` {=html} to stop adding fields): >

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

What class should this entity be related to?: > Usager

hat type of r elationship is this?:

Type Description

ManyToOne Each Commande relates to one Usager

ManyToOne

Is the Commande.usager property allowed to be null (nullable)? (yes/no) [yes]: > no

Do you want to add a new property to Usager so that you can access/update Commande objects from it - e.g. $usager->getCommandes()? (yes/no) [yes]: > yes

A new property will also be added to Usager: "commandes" (an ArrayCollection of Commande objects)

New property name (press `<return>` {=html} to stop adding fields): >

Entité LigneCommande

php bin/console make:entity Class name of the entity to create or update: > LigneCommande

New property name (press `<return>` {=html} to stop adding fields): >

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

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

New property name (press `<return>` {=html} to stop adding fields): >

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

Precision (total digits stored: e.g. 10) [10]: > 10 Scale (number of decimal digits to store: e.g. 2) [0]: > 2 Can this field be null in the database (nullable) (yes/no) [no]: > no

New property name (press `<return>` {=html} to stop adding fields): >

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

What class should this entity be related to?: > Commande

What type of relationship is this?: > ManyToOne

Is the LigneCommande.commande property allowed to be null (nullable)?
(yes/no) [no]: > no
Do you want to add a new property to Commande so that you can
access/update LigneCommande objects from it -
e.g. $commande->getLigneCommandes()? (yes/no) [yes]: > yes
A new property will also be added to Commande: "ligneCommandes" (an
ArrayCollection of LigneCommande objects)
New property name (press `<return>` {=html} to stop adding fields): >
produit Field type (enter ? to see all types) [string]: > relation
What class should this entity be related to?: > Produit
What type of relationship is this?: > ManyToOne
Is the LigneCommande.produit property allowed to be null (nullable)?
(yes/no) [no]: > no
Do you want to add a new property to Produit so that you can
access/update LigneCommande objects from it -
e.g. $produit->getLigneCommandes()? (yes/no) [yes]: > yes
A new property will also be added to Produit: "ligneCommandes" (an
ArrayCollection of LigneCommande objects)
New property name (press `<return>` {=html} to stop adding fields): >
Dans votre service src/Service/PanierService.php : o 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). o Le
contenu du panier devra être supprimé à l'issue de ce traitement. o
Cette méthode renverra en résultat l'entité Commande qui aura été créée

```
// Renvoie le contenu du panier dans le but de l'afficher
//   => un tableau d'éléments [ "produit" => un objet produit, "quantite"
=> sa quantite ]
public function getContenu() : array
{
  /* A COMPLETER */
    $tab = [];

    if(empty($this->panier)){
        return [];
    }

    foreach ($this->panier as $idProduit => $quantite){
        $produit = $this->produitRepository->find($idProduit);
        if($produit){
            $tab[] = [
```

```
                "produit" => $produit,
                "quantite" => $quantite
            ];
        }
    }
    return $tab;
}

public function panierToCommande(Usager $usager, EntityManagerInterface
$entityManager) : ?Commande
{
    $listItemsPanier = $this->getContenu();

    $commande = new Commande();
    $commande->setUsager($usager);
    $commande->setDateCreation(new \DateTime());
    $commande->setValidation(false);

    foreach ($listItemsPanier as $itemPanier){
        $ligneCommande = new LigneCommande();
        $ligneCommande->setProduit($itemPanier["produit"]);
        $ligneCommande->setQuantite($itemPanier["quantite"]);
        $ligneCommande->setPrix($itemPanier["produit"]->getPrix() *
$itemPanier["quantite"]);
        $ligneCommande->setCommande($commande);
        $commande->addLigneCommande($ligneCommande);
        $entityManager->persist($ligneCommande);
    }

    //Sauvegarde en base de données
    $entityManager->persist($commande);
    $entityManager->flush();

    $this->vider();

    return $commande;
}
```

TP6 - Sécurité

Pensez à bien définir les pages « protégées » dans la section access_control du fichier « security.yaml »... Toute la sécurité de votre application en dépend !

 Modifiez la validation du panier pour que l'usager qui passe la commande soit maintenant l'usager authentifié

 Modifiez éventuellement votre barre de navigation pour qu'elle s'adapte s'il y a ou non un usager authentifié

□ Mettez en place une page qui permet à l'usager authentifié de consulter la liste de ses commandes

1)Exécutez la commande php bin/console make:security:form-login □ Attention à bien indiquer l'entité utilisée pour l'authentification : App `\Entity`{=tex} `\Usager`{=tex}(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

2)Adapter SecurityController avec la locale

#[Route(path: '{_locale}', requirements: ['_locale' => '%app.supported_locales%'])] class SecurityController extends AbstractController { #[Route(path: '/login', name: 'app_login')] public function login(AuthenticationUtils $authenticationUtils): Response { // get the login error if there is one $error = $authenticationUtils->getLastAuthenticationError();

```
    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

    return $this->render('security/login.html.twig', [
        'last_username' => $lastUsername,
        'error' => $error,
    ]);
}

#[Route(
    path: '/logout',
    name: 'app_logout')
]
public function logout(): void
{
    throw new \LogicException('This method can be blank - it will be
intercepted by the logout key on your firewall.');
}
```

}

3. Adapter le formulaire login.html.twig

Le formulaire est en très grande partie entièrement autogénéré. C'est très simple.

4. Adapter la configuration security.yaml

Voir Cours 6 page 15

5)6)7) Utiliser l'usager authentifié

S'aider du cours 6 page 17

La méthode getUser hérité de AbstractController renvoie l'utilisateur (usager) connecté. En Twig, on utilise la méthode is_granted pour identifier un utilisateur avec un rôle défini dans le fichier de configuration de la sécurité comme ci-dessous.

public function commander(PanierService $panierService, EntityManagerInterface $entityManager): Response { $usager = $this->getUser(); $commande = $panierService->panierToCommande($usager, $entityManager);

```
 return $this->render('panier/commande.html.twig', [
      "nomUsager" => $usager->getNom(),
      "prenomUsager" => $usager->getPrenom(),
       "idCommande" => $commande->getId(),
       "date" => $commande->getDateCreation(),
```

]);

`<i class="fas fa-user">` {=html} `</i>` {=html}{% if is_granted('ROLE_CLIENT') %} {{ app.user.prenom }} {% else %} Usager {% endif %} `</span>` {=html}

```
<div class="dropdown-menu" aria-labelledby="dropdown02" style="min-width: 5rem;">
```

```
    <a class="dropdown-item {% if is_granted('ROLE_CLIENT') %} disabled {% endif %} " href="{{ path('app_login') }}">Connexion</a>
    <a class="dropdown-item {% if is_granted('ROLE_CLIENT') %} disabled {% endif %} " href="{{ path('app_usager_new') }}">Inscription</a>
    <a class="dropdown-item {% if not is_granted('ROLE_CLIENT') %} disabled {% endif %} " href="{{ path('app_usager_index') }}">Mon compte</a>
    <a class="dropdown-item {% if not is_granted('ROLE_CLIENT') %} disabled {% endif %} " href="{{ path('app_usager_commandes') }}">Mes commandes</a>
    <a class="dropdown-item {% if not is_granted('ROLE_CLIENT') %} disabled {% endif %} " href="{{ path('app_logout') }}">Déconnexion</a>
</div>
```

`</i>` {=html}

{% if is_granted('ROLE_CLIENT') and nombreProduits > 0 %}

```
<div class="col-md-6 align-self-center btn-group btn-group" role="group">
```

```
     <a class="btn btn-primary" href="{{ path('app_panier_commander') }}">
           Passer la commande
       </a>
  </div>
```

{% endif %}

TP7 - Extensions

Requête avec le QueryBuilder, contrôleur imbriqué et inclusion dans un

template

Dans src/Repository/ProduitRepository.php

public function nProduitsLesPlusVendus(int $n): array { return

$this->createQueryBuilder('p') ->select('p as produit, SUM(lc.quantite) as totalVentes') -

>join('p.ligneCommandes', 'lc') // Joindre les lignes de commande ->groupBy('p.id') // Grouper par produit -

>orderBy('totalVentes', 'DESC') // Trier par ventes décroissantes ->setMaxResults($n)

// Limiter à n résultats ->getQuery() ->getResult(); }

Dans templates/boutique/topVentes.html.twig

```
<div class="card-body text-muted text-center">
    <ul class="list-group">
        {% for queryField in prodPlusVendus %}
        <a href="{{ queryField.produit.visuel }}">
            <li class="list-group-item text-center clickable top-produit">
                <img src="/{{ queryField.produit.visuel }}" style="height:
40px;">
                <p class="top-produit">{{ queryField.produit.libelle }}<br>
<i>
                        {{ queryField.totalVentes }}
                        vente(s)</i>
                </p>
            </li>
        </a>
        {% endfor %}
    </ul>
</div>
```

Dans templates/base.html.twig

```
{{ render(controller('App\\Controller\\BoutiqueController::topVentes')) }}
```