SYMFONY6-PHP8 NOTIONS AVANCÉES

METTRE EN PLACE UN MENU DYNAMIQUE

Afin d'améliorer l' UX d'un site, nous utilisons souvent le principe de modifier l'affichage de l'élément du menu correspondant à la page où l'utilisateur se trouve actuellement.

Dans **Bootstrap** il existe une classe permettant de définir un style de mise en évidence sur un item du menu.

Lien vers la doc Bootstrap du active

Le principe ici sera de transmettre une propriété au template qui permettra ensuite d'être utilisé dans une condition au sein de notre menu de navigation. Nous appellerons cette propriété `current_menu`.

Il faudra donc sur chaque fonction au sein des controllers passer au template associé la propriété correspondante à la partie du site où l'utilisateur se trouve lorsqu'il fait appel à cette route.

Pour rappel, nous avons 3 pages blog, about et contact. Pour le dashboard le lien se trouve sur la `navbar-brand`.

Voici un exemple au sein du controller BlogController :

```php
/**
 * Permet d'afficher la liste des thèmes et des articles
 * @return Response
 */
#[Route('/blog', name: 'blog')]
public function index(): Response
{
    $articles = $this->entityManager->getRepository(Articles::class)->findAll();
    $themes = $this->entityManager->getRepository(Themes::class)->findAll();

    return $this->render( view: 'blog/index.html.twig', [
        'articles' => $articles,
        'themes' => $themes,
        'current_menu' => 'blog'
    ]);
}

/**
 * Permet d'afficher les articles par thème
 * @param $slug
 * @return Response
 */
#[Route('/blog/themes/{slug}', name: 'blog_themes')]
public function show($slug): Response
{
    $articles_themes = $this->entityManager->getRepository(Themes::class)->findOneBySlug($slug);

    return $this->render( view: 'blog/show.html.twig', [
        'articles_themes' => $articles_themes,
        'current_menu' => 'blog'
    ]);
}
```

Nous utiliserons ensuite un affichage conditionnel dans notre template twig de la navigation comme ceci :

```twig
<li class="nav-item">
    <a class="nav-link {% if current_menu == 'blog' %}active{% endif %}" href="{{ path('blog') }}">Blog</a>
</li>
```

TP 1 NAVIGATION ADMINISTRATION

Vous devrez tout simplement mettre en place le système de navigation dynamique vu précédemment au sein de la partie administration.

ATTENTION : La partie administration comporte plus de fonctions et donc de routes.

ALLER PLUS LOIN AVEC LES BLOCS

Nous avons vu que le moteur de template **Twig** permet d'utiliser un système de blocs qui permettent de créer des espaces réservés et des remplacements. Prenons l'exemple de notre `base.html.twig` , nous avons mis en place un bloc **content** qui représente le `<main>` de notre page, le contenu principal. Nous pouvons ensuite appeler ce bloc au sein de nos templates.

Il est possible de créer un bloc javascripts par exemple et ensuite l'appeler au besoin de nos templates.

Nous allons maintenant utiliser le système de blocs afin de parfaire notre blog. En effet notre structure est faite de manière à ce que la section hero est appelée sur toutes les pages. Il faudrait que cette section ne s'affiche que sur la page d'accueil du site.

Voici comment définir l'affichage d'un bloc sous condition dans notre fichier `base.html.twig`

```
<body>
    <!-- Le header du site qui contient la navigation-->
    {% if block("header") is defined %}
        {% include './partials/header.html.twig' %}
    {% else %}
        {% include './partials/nav.html.twig' %}
    {% endif %}
    <main>
        <!-- Block content avec le contenu du main -->
        {% block content %}{% endblock %}
    </main>
    <!-- Le footer du site -->
    {% include './partials/footer.html.twig' %}
    <!-- Bootstrap core Js -->
    <script src="{{ asset('assets/js/bootstrap.bundle.min.js') }}"></script>
</body>
```

Ensuite il suffit juste de définir ce block dans les templates où l'on souhaite afficher le header complet :

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

{% block title %}Accueil - My Blog{% endblock %}

{% block header %} {% endblock %}

{% block content %}
<div class="container-md mt-5 pt-5">
    <!-- about section start -->
```

ENVOYER DES EMAILS

Nous allons ici mettre en place un système de mail lors de l'inscription qui s'enverra automatiquement, nous en aurons également besoin afin de pouvoir proposer à notre utilisateur de ré initialiser son mot de passe.

Symfony embarque un composant qui s'appelle Mailer et qui simplifie l'envoi d'emails. Lorsque l'on travaille sur un projet hebergé sur un serveur nous avons accès à un mail qui permet d'avoir une adresse smtp afin d'envoyer des emails depuis une adresse type info@nom-du-site.fr

Lien vers la doc du Mailer

Le mailer de Symfony peut être utilisé également avec une boite type gmail, cependant dans un usage professionnel nous ferons le plus souvent appel à un gestionnaire d'emailing client.

Pour ce cours nous utiliserons MailJet qui fonctionne très bien avec Symfony, permet d'envoyer 6000 mails gratuitement par mois et 200 par jour.

INSTALLER MAILJET

Il va falloir dans un premier temps créer un compte MailJet. Lorsque MailJet demande le rôle il faut choisir développeur.

Lors du test d'envoi de premier mail il faut choisir API et Php comme langage.

Lien vers MailJet

Lien vers le getting started de MailJet

CONFIGURER MAILJET

CREER UN TEMPLATE

Notre compte créé, nous allons aller dans Modèles puis Mes modèles Transactionnels afin de créer un premier modèle ou template d'email. On choisit un modèle de départ et on le nomme. Nous réalisons ensuite un template avec deux variables, une title et une content qui représenteront le titre de l'email et le contenu. On voit ici tout l'intérêt de cette solution car elle permet de créer le template sans compétences de code, pratique dans le cadre d'un travail avec une équipe marketing.

De plus MailJet permet d'automatiser l'envoi d'email. Par exemple si vous demandez la date de naissance à l'utilisateur lors de son inscription vous pouvez automatiquement envoyer un email pour son anniversaire avec un coupon de réduction par exemple.

Nous pouvons ensuite aller dans les paramètres du template afin de mettre la langue en français, on peut ensuite saisir l'objet, ici Message de la part de My Blog l'adresse d'expédition sera contact@my-blog.fr. Ces valeurs ne nous servent que pour l'exemple, et afin de comprendre la mise en place.

Une fois le template créé nous pouvons le sauvegarder et le publier. Nous obtiendrons comme cela un id pour ce template que nous utiliserons plus tard.

INSTALLER MAILJET DANS NOTRE PROJET

Sur la page du getting started nous avons toute la procédure à suivre pour installer et mettre en place MailJet.

Nous utiliserons la version 3.1 de l'API.

Lien vers la doc MailJet API V3.1 template

Lien dépôt Github MailJet-apiV3-php

Nous allons donc commencer par copier la commande composer qui permet d'installer MailJet.

Plutôt que de tester l'envoi d'un premier email en copiant collant l'exemple au sein d'un controller et en rafraîchissant la page correspondante, nous allons directement utiliser notre template et mettre en place l'envoi d'email au sein de notre projet.

Dans le but d'éviter la répétition de code et d'alourdir nos controller nécessitant l'envoi de mails, nous allons créer un **service Mail** qui nous permettra de facilement générer un envoi de mail dans n'importe quel controller.

CREATION DU SERVICE MAIL

Observons l'exemple fournit par la documentation, nous voyons que nous devons instancier **l'objet MailJet** qui permettra ensuite, au sein de nos controllers, de faire appel à la méthode **send**.

Nous commencerons par ajouter deux variables d'environnement. En effet nous allons avoir besoin de deux données, l'api key de MailJet et l'api secret key de Mailjet que vous pouvez trouver dans le template d'exemple ou dans votre compte MailJet. Ces deux données étant sensibles, il faut utiliser le fichier env qui n'est pas versionné sur Github. Vous pouvez également utiliser les paramètres dans le fichier services.yaml dans le cas d'un controller qui étend la classe AbstractController

Il faudra donc utiliser la super globale `$_ENV()` afin d'accéder à nos variables d'environnement au sein de notre service.

Lien vers la doc Symfony encryptage

Nous pouvons utiliser l'exemple au sein d'une fonction **send** et remplacer les valeurs par celles souhaitées. Il faudra donc passer 4 paramètres à notre fonction send.

- **$to_email** : qui représente l'adresse email à qui on veut envoyer le mail
- **$subject** : qui représente l'objet de l'email
- **$title** : qui représente le contenu de la variable title de notre template d'email
- **$content** : qui représente le contenu de la variable content de notre template d'email

Il faudra donc préciser également dans le `From` l'adresse expéditrice et le nom de l'expéditeur. Nous devrons également mettre l'id de notre template créé dans la variable `TemplateID`.

Attention également aux différents use.

Voici le service Mail avec un dd() afin de s'assurer de l'envoi du mail :

```php
namespace App\Services;

use Mailjet\Client;
use Mailjet\Resources;

class Mail
{
    public function send($to_email, $subject, $content, $title): void
    {
        $mj = new Client($_ENV['MAILJET_API_KEY'], $_ENV['MAILJET_SECRET_KEY'], call: true, ['version' => 'v3.1']);
        $body = [
            'Messages' => [
                [
                    'From' => [
                        'Email' => "contact@xn--e-kiga-dev-c9a.fr",
                        'Name' => "My-Blog"
                    ],
                    'To' => [
                        [
                            'Email' => $to_email,
                            'Name' => 'membre de my-blog'
                        ]
                    ],
                    'TemplateID' => 3470773,
                    'TemplateLanguage' => true,
                    'Subject' => $subject,
                    'Variables' => [
                        'content' => $content,
                        'title' => $title
                    ]
                ]
            ]
        ];
        $response = $mj->post(Resources::$Email, ['body' => $body]);
        $response->success() && dd($response->getData());
    }
}
```

Nous pouvons ensuite ajouter l'envoi d'un mail de test depuis le controller de la homepage afin de vérifier que l'envoi est un succès :

```php
/**
 * Permet d'afficher la page d'accueil
 * @param EntityManagerInterface $entityManager
 * @return Response
 */
#[Route('/', name: 'homepage')]
public function index(EntityManagerInterface $entityManager): Response
{
    $articles = $entityManager->getRepository(Articles::class)->findAll();
    $themes = $entityManager->getRepository(Themes::class)->findAll();
    $mail = new Mail();
    $mail->send(
        to_email: 'gradejulien@gmail.com',
        subject: 'test',
        content: 'essai envoi',
        title: 'test mail'
    );

    return $this->render( view: 'homepage/index.html.twig', [
        'articles' => $articles,
        'themes' => $themes,
        'current_menu' => 'homepage'
    ]);
}
```

Une fois ce test fait on peut effacer le dd() et l'envoi du mail de test au sein du HomePageController. Nous allons maintenant gérer l'envoi d'un mail lors de l'inscription.

ENVOYER UN EMAIL

Nous allons maintenant gérer l'envoi du mail de bienvenue, notez que l'on pourrait ici mettre en place un email de vérification d'adresse email avec un lien vers une route qui aurait juste pour but de faire passer un champ de type booléen à true au sein de l'entité User.

Ainsi avec l'affichage conditionnel, nous pouvons afficher un message à l'utilisateur pour qu'il vérifie son email, ne pas valider d'action sans que ce champ ne soit à true. Enfin bref

une fois de plus tout est possible. Nous allons mettre en place par la suite un système de mot de passe oublié qui utilisera notre service Mail.

```php
// Si l'email n'existe pas
if(!$search_email){
    // On gère la sécurité avec l'encodage des mots de passe
    $password = $encoder->hashPassword( $user, $user->getPassword() );

    // On ré injecte le mot de passe crypté
    $user->setPassword($password);

    // On persiste les données en base(uniquement lors de la création)
    $this->entityManager->persist($user);
    // On enregistre les données en base
    $this->entityManager->flush();

    // Ici on ajoute un flash pour confirmer la création du compte.
    $this->addFlash( type: 'success', message: 'Vous êtes désormais inscrit vous pouvez vous connecter' );

    // On envoi l'email de confirmation d'inscription
    $title = 'Bienvenue sur My-Blog';
    $content = 'Merci, tu as décidé de rejoindre notre communauté, si tu veux être prévenu lors de la sortie d\'un nnouvel article, abonnes toi à la newsletter.';
    $mail = new Mail();
    $mail->send(
        $user->getEmail(),
        subject: 'Message de la part de My-Blog',
        $content,
        $title
    );

    return $this->redirectToRoute( route: 'app_login');
```

GESTION DE COMPTE

Nous allons commencer par créer une route qui aura pour but d'afficher la gestion de compte sur l'url /mon-compte, on le nommera `AccountController`.

Pour le moment nous n'aurons pas grand-chose à afficher à notre utilisateur hormis un petit texte indiquant que c'est ici qu'il peut gérer son compte et un bouton modifier mon mot de passe. Si nous avions décidé de demander plus d'infos à notre utilisateur lors de l'inscription nous pourrions les afficher ici.

Si nous avions un système de commentaires nous pourrions également offrir la possibilité de les voir dans cet espace.

Le controller :

```php
class AccountController extends AbstractController
{
    /**
     * Permet d'afficher la gestion de compte à l'utilisateur
     * @return Response
     */
    #[Route('/mon-compte', name: 'account')]
    public function index(): Response
    {
        return $this->render( view: 'account/index.html.twig', [
            'current_menu' => 'homepage',
        ]);
    }
}
```

Le template :

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

{% block title %}Mon compte - My Blog{% endblock %}

{% block content %}
<div class="container-md mt-5 pt-5">
    <h3>Gestion de compte</h3>
    <p>C'est ici que vous gérez tout ce qui vous concerne, vous pouvez notamment
    <div class="d-md-flex justify-content-md-center align-items-md-center">
        <a href="#" class="btn btn-warning">Modifier mot de passe</a>
    </div>
```

Une fois ce controller et template associé créé, nous allons ajouter de la sécurité grâce une fois de plus aux access-control du fichier security.yaml

```yaml
access_control:
    - { path: ^/admin/connexion, roles: PUBLIC_ACCESS }
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/mon-compte, roles: ROLE_USER}
    # - { path: ^/profile, roles: ROLE_USER }
```

On pourrait être tenté de créer la fonction et la route associée permettant de modifier son mot de passe dans ce controller. Cependant, toujours dans la bonne pratique de découper les responsabilités, nous allons créer un controller `AccountPasswordController`.

Il ne comportera qu'une seule route qui sera sur /mon-compte/modifier-mot-de-passe et aura pour nom account_password. Nous placerons les templates associés à la gestion de compte dans un dossier account, ici nous nommerons le template password.html.twig.

Nous pourrons ensuite mettre à jour les liens dans la barre de navigation mais aussi dans le template associé à la route /mon-compte.

Nous allons ensuite créer un formulaire grâce au maker de Symfony. On le nommera ChangePassword, il sera lié à l'entité User.

Voici le formulaire :

```
$builder
    ->add( child: 'old_password', type: PasswordType::class, [
        'label' => 'Mon mot de passe actuel',
        'mapped' => false,
        'attr' => [
            'placeholder' => 'Veuillez saisir votre mot de passe actuel',
            'class' => 'form-control'
        ]
    ])
    ->add( child: 'new_password', type: RepeatedType::class, [
        'type' => PasswordType::class,
        'mapped' => false,
        'invalid_message' => 'Le mot de passe et la confirmation doivent être identiques',
        'required' => false,
        'first_options' => [
            'label' => 'Votre nouveau mot de passe',
            'attr' => [
                'placeholder' => 'Merci de saisir votre nouveau mot de passe',
                'class' => 'form-control'
            ]
        ],
        'second_options' => [
            'label' => 'Confirmez votre nouveau mot de passe',
            'attr' => [
                'placeholder' => 'Merci de confirmer votre nouveau mot de passe',
                'class' => 'form-control'
            ]
        ],
    ])
    ->add( child: 'submit', type: SubmitType::class, [
        'label' => 'Valider les modifications',
        'attr' => [
            'class' => 'btn btn-success mx-auto mt-5'
        ]
    ])
;
```

Vous noterez que même si notre formulaire est lié à l'entité User nous avons fait le choix de créer de nouveaux champs correspondant à l'ancien mot de passe, ce qui ajoute une sécurité supplémentaire, mais aussi un champ de type répété pour la saisie du nouveau mot de passe.

Nous allons nous occuper de mettre à jour l'entité User correspondante à l'utilisateur connecté dans notre `AccountPasswordController`.

Voici le controller :

```php
#[Route('/mon-compte/modifier-mot-de-passe', name: 'account_password')]
public function index(Request $request, UserPasswordHasherInterface $encoder, EntityManagerInterface $entityManager): Response
{
    $notification = null;

    if($user = $this->getUser()){

        $form = $this->createForm( type: ChangePasswordType::class, $user);

        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid()){
            // On récupère l'ancien mot de passe saisi pour vérifier sa validité
            $old_pwd = $form->get('old_password')->getData();

            if($encoder->isPasswordValid($user, $old_pwd)) {
                // on récupère le nouveau mot de passe saisi
                $new_pwd = $form->get('new_password')->getData();
                // on encode ce nouveau mot de passe
                $password = $encoder->hashPassword($user, $new_pwd);
                // on attribue au champs password la valeur du nouveau mot de passe encodé
                /**
                 * @var User $user
                 */
                $user->setPassword($password);
                $entityManager->flush();
                $notification = "Vos informations ont bien été mises à jour";
            }else{
                $notification = "Votre mot de passe actuel n'est pas le bon";
            }
        }
    }else{
        return $this->redirectToRoute( route: 'app_login');
    }
```

Afin d'illustrer une autre méthode d'alerte nous avons utilisé un système de notification. Vous noterez l'utilisation ici du typage pour la méthode setPassword, cela évite une alerte pour polymorphic call.

Voici le template associé :

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

{% block title %}Modifier mot de passe - My Blog{% endblock %}

{% block content %}
    <div class="container-md mt-5 pt-5">
        <h3>Modifier mon mot de passe</h3>
        <p>C'est ici que vous pouvez notamment modifier votre mot de passe.</p>
        <div class="mt-5">
            {% if notification %}
                <div class="alert alert-dismissible alert-info">
                    {{ notification }}
                </div>
            {% endif %}
            {{ form(form) }}
        </div>
        <div class="row pt-md-5 mt-5 mb-3">
            <div class="col-12 my-2 col-lg-6 mx-md-auto">
                <a href="{{ path('account') }}" class="btn btn-outline-info w-100">Retour à mon compte</a>
            </div>
        </div>
    </div>
{% endblock %}
```

LOGIQUE DE MOT DE PASSE OUBLIÉ

Nous allons commencer par mettre à jour notre template login.html.twig du dossier register afin de lui ajouter deux liens un sans href pour le moment permettant d'accéder au mot de passe oublié et un second permettant de s'inscrire.

```twig
    <button class="btn btn-lg btn-primary mt-5" type="submit">
        Se connecter
    </button>
</form>
<hr />
<p class="text-center small"><a class="main-blue" href="#">Mot de passe oublié ?</a></p>
<p class="text-center main-grey">Ou souhaitez-vous <a class="main-purple" href="{{ path('register') }}">vous inscrire</a></p>
```

Afin de mettre en place la logique de mot de passe oublié nous pourrions utiliser le maker de Symfony et un bundle fournit par Symfony. Cependant nous allons ici coder notre logique nous-même afin de comprendre ce qu'il se passe.

Lien vers la doc Symfony du bundle

Nous allons ici opter pour la création d'une nouvelle entité ResetPassword qui contiendra le user, un token (une string afin d'identifier l'utilisateur) et un champs createdAt. Nous utiliserons le maker afin de générer cette entité.

Voici pour le champ user qui sera une relation avec notre entité User :

```
New property name (press <return> to stop adding fields):
> user

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

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

What type of relationship is this?
------------ ---------------------------------------------------------------------
 Type         Description
------------ ---------------------------------------------------------------------
 ManyToOne    Each ResetPassword relates to (has) one User.
              Each User can relate to (can have) many ResetPassword objects

 OneToMany    Each ResetPassword can relate to (can have) many User objects.
              Each User relates to (has) one ResetPassword

 ManyToMany   Each ResetPassword can relate to (can have) many User objects.
              Each User can also relate to (can also have) many ResetPassword objects

 OneToOne     Each ResetPassword relates to (has) exactly one User.
              Each User also relates to (has) exactly one ResetPassword.
------------ ---------------------------------------------------------------------

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne

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

Do you want to add a new property to User so that you can access/update ResetPassword objects from it - e.g. $user->getResetPasswords()? (yes/no) [yes]:
> no

updated: src/Entity/ResetPassword.php
```

Et pour le champ token et createdAt :

```
Add another property? Enter the property name (or press <return> to stop adding fields):
> token

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

Field length [255]:
>

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

updated: src/Entity/ResetPassword.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> createdAt

Field type (enter ? to see all types) [datetime_immutable]:
>

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

updated: src/Entity/ResetPassword.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>


 Success!
```

Nous pouvons ensuite créer notre controller ResetPasswordController la fonction index sera sur la route /mot-de-passe-oublie et aura pour nom reset_password. Nous pourrons ensuite mettre à jour le href du template login.html.twig.

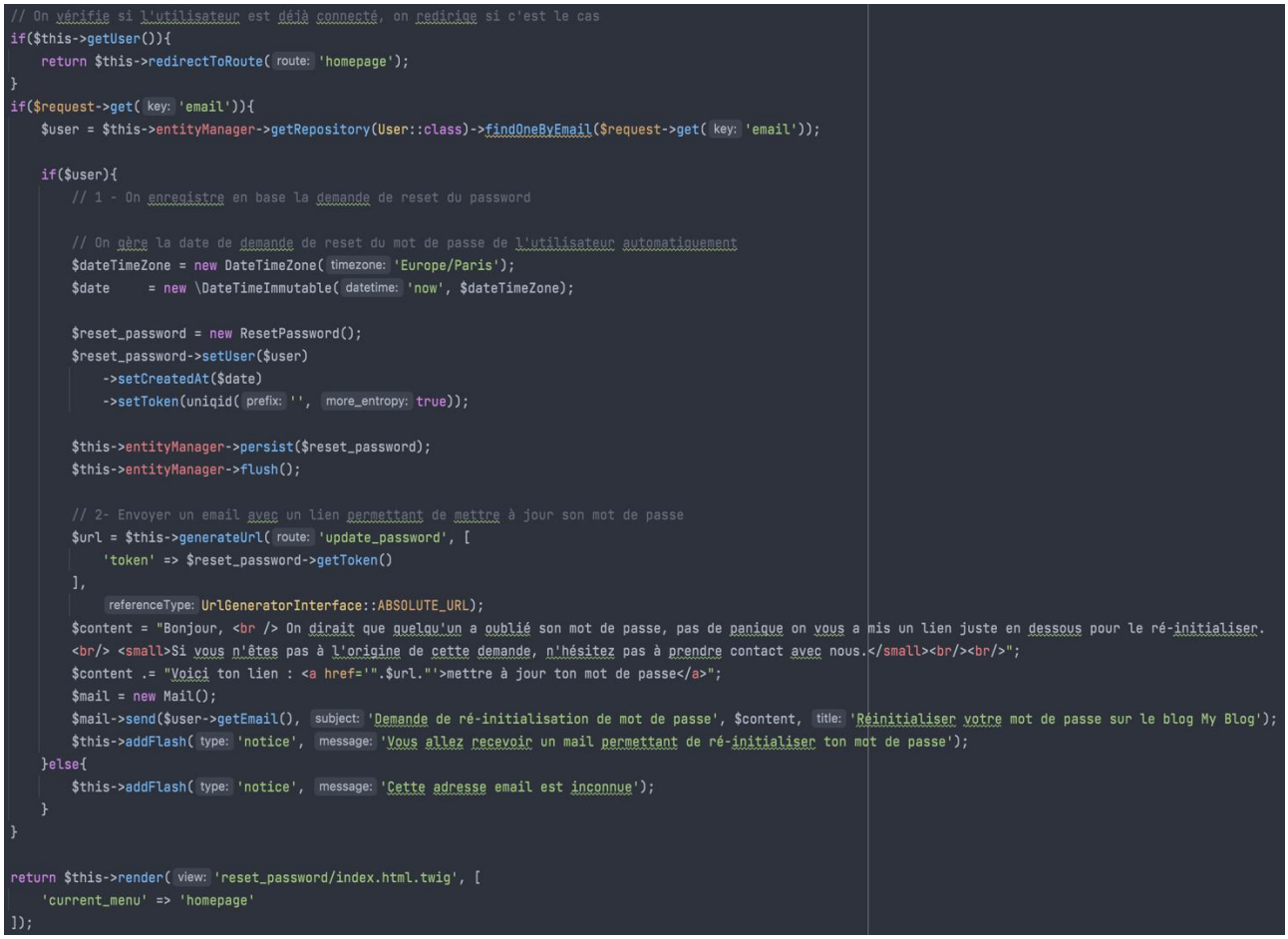
Le template associé ressemblera à celui login.html.twig que nous allons modifier comme suit :

```twig
{% block content %}
    <section>
        <div class="container-md mt-5 mx-auto">
            <div class="w-50 mx-auto">
                {% for message in app.flashes('notice') %}
                    <div class="alert alert-info">
                        {{ message }}
                    </div>
                {% endfor %}
                <form method="post">
                    <h1 class="h3 mb-3 main-grey font-weight-normal">My Blog - Mot de passe perdu</h1>
                    <label for="inputEmail">Votre email</label>
                    <input type="email" name="email" id="inputEmail" class="form-control" autocomplete="email" required autofocus>

                    {#
                    Uncomment this section and add a remember_me option below your firewall to activate remember me functionality.
                    See https://symfony.com/doc/current/security/remember_me.html
                    <div class="checkbox mb-3">
                        <label>
                            <input type="checkbox" name="_remember_me"> Remember me
                        </label>
                    </div>
                    #}

                    <button class="btn mt-5 btn-lg px-5 btn-success" type="submit">
                        Réinitialiser mon mot de passe
                    </button>
                    <hr />
                    <p class="text-center main-grey">Ou souhaitez-vous <a class="main-purple" href="{{ path('app_login') }}">vous connecter</a></p>
                </form>
            </div>
        </div>
    </section>
{% endblock %}
```

Nous allons pouvoir écrire notre fonction index qui devra nous permettre de vérifier que l'email saisit existe en base de données et de créer l'entrée dans la table reset_password en générant un token, une date de création et en établissant une relation avec le User correspondant à l'email saisit.

Il faudra aussi mettre en place des notifications, ainsi que de faire l'envoi du mail contenant le lien que nous devrons générer.

Voici cette fonction :



```php
// On vérifie si l'utilisateur est déjà connecté, on redirige si c'est le cas
if($this->getUser()){
    return $this->redirectToRoute( route: 'homepage');
}
if($request->get( key: 'email')){
    $user = $this->entityManager->getRepository(User::class)->findOneByEmail($request->get( key: 'email'));

    if($user){
        // 1 - On enregistre en base la demande de reset du password

        // On gère la date de demande de reset du mot de passe de l'utilisateur automatiquement
        $dateTimeZone = new DateTimeZone( timezone: 'Europe/Paris');
        $date     = new \DateTimeImmutable( datetime: 'now', $dateTimeZone);

        $reset_password = new ResetPassword();
        $reset_password->setUser($user)
            ->setCreatedAt($date)
            ->setToken(uniqid( prefix: '', more_entropy: true));

        $this->entityManager->persist($reset_password);
        $this->entityManager->flush();

        // 2- Envoyer un email avec un lien permettant de mettre à jour son mot de passe
        $url = $this->generateUrl( route: 'update_password', [
            'token' => $reset_password->getToken()
        ],
             referenceType: UrlGeneratorInterface::ABSOLUTE_URL);
        $content = "Bonjour, <br /> On dirait que quelqu'un a oublié son mot de passe, pas de panique on vous a mis un lien juste en dessous pour le ré-initialiser.
        <br/> <small>Si vous n'êtes pas à l'origine de cette demande, n'hésitez pas à prendre contact avec nous.</small><br/><br/>";
        $content .= "Voici ton lien : <a href='".$url."'>mettre à jour ton mot de passe</a>";
        $mail = new Mail();
        $mail->send($user->getEmail(), subject: 'Demande de ré-initialisation de mot de passe', $content, title: 'Réinitialiser votre mot de passe sur le blog My Blog');
        $this->addFlash( type: 'notice', message: 'Vous allez recevoir un mail permettant de ré-initialiser ton mot de passe');
    }else{
        $this->addFlash( type: 'notice', message: 'Cette adresse email est inconnue');
    }
}

return $this->render( view: 'reset_password/index.html.twig', [
    'current_menu' => 'homepage'
]);
```

Nous pouvons à ce stade tester que le mail part bien, notre lien fonctionne, il nous reste à mettre en place la fonction `update` qui sera sur la route /modifier-mot-de-passe/{token} et portera le nom `update_password`.

Cette route devra renvoyer un formulaire que l'on nommera `ResetPasswordType` que nous créerons grâce au maker le Type se rajoute automatiquement. Ce formulaire ne se basera sur aucune entité mais reprendra le formulaire `ChangePasswordType`.

Voir le formulaire page suivante.

```php
$builder
    ->add( child: 'new_password', type: RepeatedType::class, [
        'type' => PasswordType::class,
        'invalid_message' => 'Le mot de passe et la confirmation doivent être identiques',
        'required' => true,
        'first_options' => [
            'label' => 'Votre nouveau mot de passe',
            'attr' => [
                'placeholder' => 'Merci de saisir votre nouveau mot de passe',
                'class' => 'form-control'
            ]
        ],
        'second_options' => [
            'label' => 'Confirmez votre nouveau mot de passe',
            'attr' => [
                'placeholder' => 'Merci de confirmer votre nouveau mot de passe',
                'class' => 'form-control'
            ]
        ],
    ])
    ->add( child: 'submit', type: SubmitType::class, [
        'label' => 'Valider les modifications',
        'attr' => [
            'class' => 'btn btn-success w-50 mx-auto mt-5'
        ]
    ])
;
```

La fonction update page suivante.

```php
#[Route('/modifier-mot-de-passe/{token}', name: 'update_password')]
public function update($token, Request $request, UserPasswordHasherInterface $encoder): Response
{
    // On récupère le reset password associé au token
    $reset_password = $this->entityManager->getRepository(ResetPassword::class)->findOneByToken($token);

    if(!$reset_password){
        return $this->redirectToRoute( route: 'reset_password');
    }

    // Vérifier si le createdAt = now - 3h
    $now = new \DateTimeImmutable();
    if($now > $reset_password->getCreatedAt()->modify('+3 hour')){
        // Cas ou le token a expiré 3h de validité
        $this->addFlash( type: 'notice', message: 'Ta demande de ré-initialisation de mot de passe a expirée. Merci de renouveler la demande');
        return $this->redirectToRoute( route: 'reset_password');
    }

    // On créé le formulaire de modification de mot de passe
    $form = $this->createForm( type: ResetPasswordType::class);
    // On récupère les données du formulaire dans la requête
    $form->handleRequest($request);
    // On vérifie si le formulaire est valide et si il est bien soumit
    if($form->isSubmitted() && $form->isValid()){
        // On récupère les données du champs saisi correspondant au nouveau mot de passe
        $new_pwd = $form->get('new_password')->getData();
        // On encode ce nouveau mot de passe
        $password = $encoder->hashPassword($reset_password->getUser(), $new_pwd);
        // On enregistre le mot de passe encodé
        $reset_password->getUser()->setPassword($password);
        // On enregistre les données en base
        $this->entityManager->flush();
        // Notification utilisateur
        $this->addFlash( type: 'success', message: 'Mot de passe enregistré');
        // On redirige l'utilisateur
        return $this->redirectToRoute( route: 'app_login');
    }
    return $this->render( view: 'reset_password/update.html.twig', [
        'form' => $form->createView(),
        'current_menu' => 'homepage'
    ]);
}
```

Nous pouvons maintenant tester notre fonctionnalité, il faudra penser à ajouter le bout de code permettant d'afficher le flash success sur la page login.html.twig.

LES CONTRAINTES DE VALIDATION

Nous avons brièvement abordé cette notion lors du cours précédent lorsque nous voulions nous assurer que le titre soit unique pour nos themes et nos articles.

Nous allons maintenant voir en détail cette notion et sa mise en place. En effet, il ne faut jamais se fier à ce que l'utilisateur va saisir dans un formulaire, de plus ils représentent les principaux points d'entrée des visiteurs malveillants, injection de script permettant de récupérer des données par exemple. C'est pourquoi il est toujours nécessaire d'effectuer des vérifications sur les données saisies côté serveur et de coupler ça avec une vérification côté front-end grâce au typage des input ou à l'utilisation d'un langage de programmation.

Symfony nous fournit l'outil parfait pour valider nos formulaires, les contraintes de validation avec le **ValidatorInterface**. Ce composant va nous offrir une multitude de contraintes.

[Lien vers la doc Symfony Validation]

Dans le cadre de notre projet de blog, la plupart des saisies via formulaire se font par l'administrateur ce qui semble comporter moins de risque. Cependant les contraintes de validation permettent également d'afficher les messages d'erreurs et donc de sensiblement améliorer l'UX du site.

Nous allons commencer par ajouter des contraintes de validation sur notre entité themes.

Par convention afin d'utiliser le composant de validation de Symfony nous utilisons un alias comme vu dans la documentation.

```
use Symfony\Component\Validator\Constraints as Assert;
```

Sur cette entité nous souhaitons que le champs title soit obligatoirement saisit, qu'il comporte au minimum 5 caractères et au maximum 50. Nous devrons afficher les messages correspondants. Notez que ce champ étant de type string il n'autorise de toute façon par nature pas plus de 255 caractères.

De plus, notre formulaire construit avec Symfony et les form fields nous assure du type de champs attendu, pour le title un TextType. Il est tout à fait possible de créer ses validations au sein des formulaires mêmes, cependant cela alourdit sensiblement les formulaires et rend les choses plus complexes à maintenir.

Il est également possible de configurer ses contraintes dans le fichier validator.yaml du dossier config.

Nous utiliserons les attributs afin de rédiger nos contraintes au sein même des entités, il est possible également d'utiliser les annotations, pour respecter les bonnes pratiques de la version 8 de php nous faisons le choix des attributs.

Voici comment rédiger les contraintes sur le champ title :

```php
/**
 * @ORM\Column(type="string", length=255)
 */
#[Assert\NotBlank]
#[Assert\Length(
    min: 5,
    max: 50,
    minMessage: 'Le titre doit comporter au moins 5 caractères',
    maxMessage: 'Le titre ne peut pas excéder 50 caractères',
)]
private ?string $title;
```

Sur cette entité nous n'aurons pas plus de contraintes à mettre en place, le champs illustration sera traité dans un prochain chapitre, le champs slug est saisit automatiquement par notre service. Nous pouvons ajouter des contraintes sur nos autres entités.

Il existe de très nombreuses contraintes qui permettent de s'adapter à tous les types de besoins.

Lien vers la doc Symfony Les contraintes

UPLOAD D'IMAGES

Nous allons maintenant traiter notre champs illustration sur l'entité themes, actuellement c'est une simple url menant à une image. Nous allons maintenant mettre en place un système d'upload et d'optimisation des images.

Nous utiliserons deux bundles pour cela VichUploader et LiipImaginebundle.

POURQUOI CHOISIR UN BUNDLE ?

Nous aurions pu faire le choix de réaliser l'upload et le traitement d'images en codant des classes php, ou utiliser un outil comme glyde afin d'optimiser les images dans des requêtes http. Cependant l'utilisation de ces deux bundles vous donnent accès à de nombreuses

fonctionnalités, comme la gestion du cache automatique, la création de filtres en quelques lignes, un composant avec des options customisables.

Tout cela se traduit par un énorme gain de temps de développement, un gain de performance. Les deux bundles que nous allons utiliser sont les plus utilisés avec Symfony ce qui facilite la maintenabilité.

VIICHUPLOADER

Comme son nom l'indique ce bundle se charge de réaliser l'upload des fichiers, il prend en charge l'orm Doctrine et permet juste en précisant les chemins de destinations de gérer l'upload. Nous pourrons ensuite utiliser directement notre formbuilder afin de rendre un champ uploadable avec Vich.

LIIPIMAGINEBUNDLE

Ce bundle s'occupe de traiter des fichiers médias, de les compresser, les éditer et faciliter la gestion du cache. Il utilise un système de filtre que l'on créé dans le fichier de config, ensuite nous pouvons dans Twig utiliser un pipe pour sélectionner le filtre souhaité.

L'UPLOAD AVEC VICH

Comme pour tout bundle que vous serez amené à utiliser sur un framework, il va falloir éplucher la documentation et la suivre pour la mise en place.

Lien vers le dépôt Git de Vich

Nous allons commencer par installer le bundle dans notre projet en utilisant composer et nous répondrons oui à la demande d'exécution de la recette.

```
➜  my_blog git:(master) composer require vich/uploader-bundle
```

Lien vers la section installation

Nous avons donc maintenant dans le dossier packages du dossier config un fichier `vich_uploader.yaml`. C'est dans ce fichier que nous allons mettre ce que l'on appelle le mapping.

Nous allons utiliser le fichier services.yaml afin d'ajouter un paramètre qui représentera le chemin d'accès à nos images.

Une fois cela fait, nous pouvons créer le mapping dans le fichier vich_uploader.yaml comme la documentation du bundle l'indique.

Lien vers la section usage du bundle

Vous noterez l'utilisation des % afin d'appeler notre paramètre du fichier services.yaml, kernel.project_dir veut dire à la racine du projet car nos images doivent se trouver dans le dossier public.

La fonctionnalité namer permet de s'assurer que chaque image aura un nom unique en ajoutant des caractères au nom du fichier, les options sur le delete sont explicites.

Nous allons maintenant devoir modifier notre entité afin de mettre en place vich. La documentation nous indique que nous allons avoir besoin de nouveaux champs au sein de notre entité.

Pour notre usage il nous faudra un champ imageFile qui représentera le fichier et un champ updatedAt qui représentera la date de modification du fichier. Nous n'allons pas utiliser le maker mais ajouter nos propriétés manuellement, nous effectuerons la migration ensuite.

Il faudra donc penser à ajouter les getters et setters de ces propriétés, une fois de plus il suffit de lire la documentation. On va commencer par préciser que cette entité comporte un champ uploadable avec Vich. Ensuite on ajoute nos champs en utilisant les annotations cette fois car les attributs posent actuellement un problème sur le cycle d'événement de l'entité.

Il suffit de modifier l'exemple de la doc avec nos valeurs, themes_images et illustration.

Voir l'ajout du champ imageFile page suivante.

```php
/**
 * @Assert\File(
 *     maxSize = "2048k",
 *     mimeTypes = {"image/png", "image/jpeg", "image/pjpeg", "image/svg+xml"},
 *     mimeTypesMessage = "Format d'image supportée : .jpg, .png, .jpeg, .svg"
 * )
 * @Vich\UploadableField(mapping="themes_images", fileNameProperty="illustration")
 *
 * @var File|null
 */
private ?File $imageFile;

/**
 *
 * @param File|null $imageFile
 */
public function setImageFile(?File $imageFile = null): void
{
    $this->imageFile = $imageFile;

    if (null !== $imageFile) {
        // It is required that at least one field changes if you are using doctrine
        // otherwise the event listeners won't be called and the file is lost
        $this->updatedAt = new \DateTimeImmutable();
    }
}

public function getImageFile(): ?File
{
    return $this->imageFile;
}
```

Voici l'ajout du champ updatedAt :

```php
/**
 * @ORM\Column(type="datetime")
 *
 * @var DateTimeInterface|null
 */
private ?DateTimeInterface $updatedAt;

/**
 * @param DateTimeInterface|null $updatedAt
 */
public function setUpdatedAt(?DateTimeInterface $updatedAt): void
{
    $this->updatedAt = $updatedAt;
}

/**
 * @return DateTimeInterface|null
 */
public function getUpdatedAt(): ?DateTimeInterface
{
    return $this->updatedAt;
}
```

Vous noterez l'ajout de contraintes de validation sur le champ imageFile.

ATTENTION Il faut être vigilant sur le typage de nos propriétés afin d'éviter les erreurs.

Le champ illustration :

```
/**
 * @ORM\Column(type="string", length=255)
 */
private ?string $illustration = null;
```

Il faut maintenant modifier notre formulaire afin d'utiliser le type de champs souhaité.

Enfin il faut modifier les templates dans lesquels nous faisons appel aux images.

Nous pouvons tester la création d'un nouveau thème.

OPTIMISER LES IMAGES AVEC LIIP

Lien vers la doc de LiipImagineBundle

Ce bundle va nous permettre d'optimiser les images en terme de taille, de poids et de chargement. Au final nous aurons un gros gain de performance et nous nous assurerons que l'image uploadé ne causera pas de problème de performance.

Nous allons commencer par suivre la documentation et installer le bundle dans notre projet.

```
➜  my_blog git:(master) composer require liip/imagine-bundle
```

Une fois l'installation du package faite et la recette acceptée, la console nous affiche les prochaines étapes.

```
Configure your transformations:
  1. You MUST verify and uncomment the configuration in config/packages/liip_imagine.yaml.
  2. You MAY configure your image transformation library (gmagick, imagick, or gd [default]).
  3. You MAY define custom transformation definitions under the filter_sets option.
```

Il nous indique ici qu'il faut mettre en place la configuration dans le fichier config/packages/liip_imagine.yaml, qu'il faut ensuite sélectionner une librairie de transformation d'images (ce qui veut dire que cette libraire devra être installée sur le serveur de production). Nous garderons gd qui est le choix par défaut et qui est très souvent installé sur les hébergements. Enfin on nous dit que l'on va devoir mettre en place un filtre afin de définir la transformation que l'on souhaite à nos images.

Nous allons nous rendre dans le fichier liip_imagine.yaml et saisir nos filtres, nous utiliserons deux filtres un miniature et un full afin d'avoir deux cas de figures différents, mais on peut créer autant de filtre que nécessaire.

[Lien vers la doc sur les filtres]

```yaml
# Documentation on how to configure the bundle can be found at: https://symfony.com/doc/current/bundles/LiipImagineBundle/basic-usage.html
liip_imagine:
    # valid drivers options include "gd" or "gmagick" or "imagick"
    driver: "gd"
    filter_sets:
        miniature:
            quality: 70
            filters:
                thumbnail:
                    size: [ 300, 200 ]
                    mode: outbound
        full:
            quality: 85
            filters:
                scale:
                    dim: [ 1200,1200 ]
```

Enfin nous n'avons plus qu'à faire appel aux filtres dans Twig afin d'afficher nos images issues de la base en utilisant vich et liip. Nous pouvons en profiter pour remettre à jour notre base et créer des articles et des thèmes en utilisant des images uploadées cette fois et non plus des url.

```twig
<!-- last themes start -->
{% for last_theme in themes|sort((a, b) => b.id <=> a.id)|slice(0,1) %}
<div class="col-lg-8 col-sm-12">
    <div class="about_img"><img alt="about me" class="w-75 mx-auto img-fluid" src="{{ vich_uploader_asset(last_theme, 'imageFile')|imagine_filter('full') }}"></div>
    <div class="like_icon"><img alt="like" src="{{ asset('./assets/img/like-icon.png') }}"></div>
    <h2 class="most_text">{{ last_theme.title }}</h2>
    <p class="lorem_text">{{ last_theme.articles|length }} Articles</p>
    <div class="social_icon_main">
        <div class="social_icon">
            <ul class="list-unstyled d-flex justify-content-center">
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/fb-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/twitter-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/instagram-icon.png') }}"></a></li>
            </ul>
        </div>
        <div class="read_bt"><a href="#">Read More</a></div>
    </div>
</div>
{% endfor %}
<!-- last theme end -->
```

GÉRER LES VUES ERREURS

Comme tout projet web, nous allons devoir mettre en place la gestion des vues erreurs, la plus célèbre étant la page 404. Symfony propose un système permettant de gérer les vues erreurs en toute simplicité.

Prenons un exemple tapons une url qui n'existe pas sur notre projet, comme dans notre fichier env est définit en environnement de développement nous avons le message d'erreur Symfony en production nous aurions une page 404 par défaut.

Si nous passons cet envrionnement en prod et que nous nettoyons le cache, nous aurons alors le rendu en production.

```
# Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
# https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration

###> symfony/framework-bundle ###
APP_ENV=prod
APP_SECRET=a7f53233119c65ba9d3a1db29c29cf3c
###< symfony/framework-bundle ###
```

ATTENTION A chaque fois que l'on bascule de l'environnement dev à prod il faut lancer la commande :

```
➜  my_blog git:(master) symfony console cache:clear
```

Nous nous assurons que le twig-pack est installé :

```
➜  my_blog git:(master) ✗ composer require symfony/twig-pack
```

Ensuite nous devons créer une structure de dossier précise dans le dossier template. Il nous faut un dossier **bundles** qui contient un dossier **TwigBundle** qui contient lui-même un dossier **Exception.**

Dans ce dossier Exception nous créons un fichier error.html.twig qui étendra de base.html.twig et qui affichera notre page d'erreur. Pensez à bien vider le cache pour voir les modifications apportées.

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

{% block title %}Page introuvable - My Blog{% endblock %}

{% block content %}
    <div class="container-md">
        <section class="m-2 m-md-5 p-2 p-md-5">
            <h3>Page introuvable</h3>
            <p>On dirait que vous vous êtes perdu</p>
            <a href="{{ path('homepage') }}" class="btn btn-outline-info">Revenir sur l'accueil du blog</a>
        </section>
    </div>
{% endblock %}
```

Notez qu'il est possible de pousser cette logique plus loin en nommant les fichiers error404.html.twig pour les erreurs ainsi de suite...

En utilisant juste un fichier error.html.twig ce sera cette unique page qui s'affichera lorsqu'une erreur de tous types se présentera.

TP 2 AGRÉMENTER LES USERS

Nous allons ajouter des champs à notre entité User, pour le moment nous n'avons qu'un **email, un mot de passe** et le **role**. Nous allons ajouter un champ **avatar** qui sera une image uploadable et obligatoire. Nous ajouterons également un **champ** pseudo qui sera une string obligatoire.

ATTENTION : Il faudra mettre à jour nos données en base et dans nos fixtures afin d'ajouter ces champs si besoin. Il faudra aussi mettre à jour notre formulaire d'inscription au site.

Le but ici sera d'ajouter l'affichage de ces informations à la vue /mon-compte. Le système de pseudo et d'avatar pourra être utilisé plus tard si l'on souhaite mettre en place un système de commentaires par exemple.

MISE EN FORME DES FORMULAIRES

LE FORMULAIRE DES ARTICLES

Avec l'ajout de nos nouvelles propriétés sur les articles, on se rend compte qu'en terme d'UX il serait intéressant de pouvoir améliorer le rendu de notre formulaire.

Jusque là nous utilisions juste l'appel au formulaire qui comportait les classes **Bootstrap** nécessaires à sa mise en forme minimum. Nous allons ici traiter le style des champs qui ne nous conviennent pas.

Lien vers la doc customisation formulaires

```
{% block content %}
    <div class="container-md mt-5">
        <div class="row d-flex justify-content-between align-items-center">
            <div class="col-md-8">
                <h2 class="my-5">Création d'un nouvel article</h2>
            </div>
            <div class="col-md-4">
                <a class="btn btn-outline-info" href="{{ path('admin_articles') }}">Retour aux articles</a>
            </div>
        </div>
        <div class="mt-5">
            {{ form_start(form) }}
                <div class="custom-control custom-radio">
                    <label class="">{{ form_label(form.themes) }}</label>
                    {{ form_widget(form.themes) }}
                </div>
                <div class="custom-control mt-3">
                    <div class="d-flex justify-content-start align-items-center">
                        <label class="mr-3">{{ form_label(form.isPrivate) }}</label>
                        {{ form_widget(form.isPrivate) }}
                    </div>
                </div>
                {{ form_row(form.title) }}
                {{ form_row(form.content) }}
            {{ form_widget(form) }}
            {{ form_end(form) }}
        </div>
    </div>
{% endblock %}
```

Vous noterez l'utilisation de `form_widget(form)` avant la fin du formulaire, en effet ceci vous assure que tout le contenu du formulaire soit affiché, même ce que vous n'avez pas traité individuellement.

AMÉLIORER L'UX GRÂCE AUX THÈMES

Nous allons améliorer l'UX de nos formulaires de manière globale, en précisant à Twig que l'on va utiliser un thème pour les formulaires.

Lien vers la doc des form thèmes

Il va donc falloir agir sur le fichier qui se trouve dans `config/packages/twig.yaml`

```yaml
twig:
    default_path: '%kernel.project_dir%/templates'
    form_themes: [ 'bootstrap_5_layout.html.twig' ]

when@test:
    twig:
        strict_variables: true
```

En ajoutant ce thème, l'affichage des erreurs se fera à la façon de Bootstrap. On peut aller encore plus loin et créer ses propres thèmes.

Le but ici est de comprendre que l'on reste libre de l'affichage de notre formulaire, nous pouvons ainsi créer des sections au sein du formulaire.

Exemple page suivante

```
{{ form_start(form) }}
    <div class="row">
        <div class="col">
            <div class="alert alert-light">
                <h2>Informations générales</h2>
                <hr />
                {{ form_row(form.title) }}
                {{ form_row(form.slug) }}
                {{ form_row(form.price) }}
                {{ form_row(form.rooms) }}
            </div>
            <div class="alert alert-light">
                <h2>Détails de l'annonce</h2>
                <hr />
                {{ form_row(form.introduction) }}
                {{ form_row(form.content) }}
            </div>
        </div>
        <div class="col">
            <div class="alert alert-light">
                <h2>Images de l'annonce</h2>
                <hr />
                {{ form_row(form.coverImage) }}
                {{ form_row(form.images) }}
            </div>
        </div>
    </div>

    <div class="alert alert-primary clearfix">
        <h2 class="alert-heading">Enregistrer mon annonce</h2>
        <p>Vous êtes sur le point de créer votre annonce,
            pas d'inquiétude vous pourrez la modifier à tout instant !</p>
        <button type="submit" class="btn btn-secondary float-right">
            <i class="fas fa-check"></i>
            Créer la nouvelle annonce</button>
    </div>

{{ form_end(form) }}
```

TP3 DES ARTICLES PRIVÉS

Vous allez devoir mettre en place un système d'articles privés qui fera en sorte que lors de la création d'un article on puisse décider qu'il ne soit pas public mais nécessite d'être inscrit sur le blog.

Pour se faire vous devrez :

- Modifier l'entité Articles
- Modifier le formulaire saisi
- Utiliser un affichage conditionnel qui affichera l'article si on est connecté et qui proposera l'inscription si on ne l'est pas, il faudra prendre en compte que si l'article n'est pas privé on l'affiche et que s'il ne l'est pas on affichera soit l'article si connecté ou l'inscription si on ne l'est pas.

TP 4 DÉTAIL DES ARTICLES

Nous allons maintenant revoir la mise en forme de nos articles. En effet si l'article possède un contenu très long, il serait intéressant de mettre en place un système qui tronque le texte.

Ainsi le bouton Read More prendrait tout son sens et mènerait sur la vue du détail d'un article. Si vous n'aviez pas encore mit en place ce bouton, il faudra le mettre.

Vous allez donc devoir mettre en place un système qui permet au sein de votre template **twig** de tronquer le texte à partir de 1000 caractères et remplacer le texte tronqué par une chaîne de caractère de votre choix '...' par exemple. Il va donc ensuite falloir créer une nouvelle route qui permettra d'afficher les détails d'un article avec une url propre.

TP5 AJOUTER UNE SUGGESTION D'ARTICLES SUR LE MÊME THÈME

Nous allons mettre en place en dessous de la vue article détail que nous venons de créer une suggestion des autres articles SI il y en a et les afficher sous forme de petites card.

Nous proposerons quatre autres articles à notre utilisateur de ce thème, il faudra exclure des propositions l'article qui est affiché dans la vue détail, ce qui veut dire que dans de rares cas nous n'aurons que 3 articles affichés.

Vous allez donc devoir ajouter un peu de logique au controller afin d'ajouter les données nécessaires et utiliser Twig ensuite pour obtenir le rendu désiré.

PAGINATION

Nous allons maintenant nous concentrer sur notre page /blog, cette page à l'heure actuelle affiche tous les articles du blog. Lorsque le blog aura un nombre d'articles conséquents l'UX ainsi que les performances en seront affectés.

Nous allons donc devoir mettre en place une pagination et nous l'utiliserons pour cette route. Il faudra envisager les choses sous forme de service afin de pouvoir utiliser cette pagination sur l'entité de notre choix et donc au sein de n'importe quel controller par la suite.

Nous pourrions envisager l'utilisation d'un bundle, cependant Doctrine permet d'effectuer des requêtes avec des options qui permettent de mettre en place une limit, un offset, bref

tout ce qui est nécessaire à paginer nos données. Nous allons donc réaliser cette pagination sans utiliser de bundle.

Commençons par créer notre service ainsi que les propriétés dont nous aurons besoin avec les getters et setters associés, nous préciserons un constructeur.

Voici pour les propriétés entityClass, orderBy, limit que l'on définit à 5, currentPage, qui par défaut vaut 1, et entityManager :

```php
// Service permettant la pagination d'une entité

/**
 * Le nom de l'entité sur laquelle on veut effectuer une pagination
 *
 * @var string
 */
private string $entityClass;

/**
 * représente le orderby de la requête
 * @var array
 */
private array $orderBy;


/**
 * Le nombre d'enregistrement à récupérer
 *
 * @var int
 */
private int $limit = 5;

/**
 * La page sur laquelle on se trouve actuellement
 *
 * @var int
 */
private int $currentPage = 1;

/**
 * Le nom de l'entité sur laquelle on veut effectuer une pagination
 *
 * @var string
 */
private $entityManager;
```

Enfin pour les propriétés twig qui permettra de rendre du html, route la route sur laquelle on pagine et templatePath qui est le chemin menant à notre template :

```php
/**
 * Le moteur de template Twig qui va permettre de générer le rendu de la pagination
 *
 * @var Environment
 */
private $twig;

/**
 * Le nom de la route que l'on veut utiliser pour les boutons de la navigation
 *
 * @var string
 */
private $route;

/**
 * Le chemin vers le template qui contient la pagination
 *
 * @var string
 */
private $templatePath;
```

Voici le constructeur :

```php
/**
 * Pagination constructor.
 * @param EntityManagerInterface $entityManager
 * @param $templatePath
 * @param RequestStack $request
 * @param Environment $twig
 */
public function __construct(EntityManagerInterface $entityManager, $templatePath, RequestStack $request, Environment $twig)
{
    $this->route           = $request->getCurrentRequest()->attributes->get( key: '_route');
    $this->entityManager   = $entityManager;
    $this->templatePath    = $templatePath;
    $this->twig            = $twig;
}
```

Je vous passe la syntaxe des getters et des setters qui sont de simples méthodes.

Nous allons maintenant écrire notre première fonction qui devra compter le nombre de page d'une entité :

```php
/**
 * Permet de récupérer le nombre de pages qui existent sur une entité particulière
 *
 * Elle se sert de Doctrine pour récupérer le repository qui correspond à l'entité que l'on souhaite
 * paginer (voir la propriété $entityClass) puis elle trouve le nombre total d'enregistrements grâce
 * à la fonction findAll() du repository
 *
 * @throws Exception si la propriété $entityClass n'est pas configurée
 * @throws \Exception
 *
 * @return int
 */
public function getPages():int
{

    if(empty($this->entityClass)){
        throw new \Exception( message: "Vous n'avez pas spécifié l'entité sur laquelle vous souhaitez
        paginer. Utilisez la méthode setEntityClass() de votre objet PaginationService !");
    }

    // On calcule l'offest
    $offset = $this->currentPage * $this->limit - $this->limit;
    // Connaitre le total des enregistrements de la table
    $repo = $this->entityManager->getRepository($this->entityClass);
    $total = count($repo->findBy([], $this->orderBy,  limit: 10000,  offset: 0));

    // On calcule le nb total de page et ceil fonction php arrondi au dessus en cas de 2,3 par ex
    // Faire la division, l'arrondi et le renvoyer
    $pages = ceil( num: $total / $this->limit);

    return $pages;
}
```

Nous allons ensuite avoir besoin d'une fonction pour récupérer nos données paginées :

```php
/**
 * Permet de récupérer les données paginées pour une entité spécifique
 *
 * Elle se sert de Doctrine afin de récupérer le repository pour l'entité spécifiée
 * puis grâce au repository et à sa fonction findBy() on récupère les données dans une
 * certaine limite et en partant d'un offset
 *
 * @throws Exception si la propriété $entityClass n'est pas définie
 * @throws \Exception
 *
 * @return array
 */
public function getData(): array
{

    if(empty($this->entityClass)){
        throw new \Exception( message: "Vous n'avez pas spécifié l'entité sur laquelle vous souhaitez
        paginer. Utilisez la méthode setEntityClass() de votre objet PaginationService !");
    }

    // On calcule l'offest
    $offset = $this->currentPage * $this->limit - $this->limit;

    // Demander au repository de trouver les éléments
    $repo = $this->entityManager->getRepository($this->entityClass);
    $data = $repo->findBy([], $this->orderBy, $this->limit, $offset);
    // Envoyer les éléments
    return $data;
}
```

Enfin une fonction display qui se chargera de l'affichage de notre pagination :

```php
/**
 * Permet d'afficher le rendu de la navigation au sein d'un template twig !
 *
 * On se sert ici de notre moteur de rendu afin de compiler le template qui se trouve au chemin
 * de notre propriété $templatePath, en lui passant les variables :
 * - page  => La page actuelle sur laquelle on se trouve
 * - pages => le nombre total de pages qui existent
 * - route => le nom de la route à utiliser pour les liens de navigation
 *
 * Attention : cette fonction ne retourne rien, elle affiche directement le rendu
 *
 * @throws LoaderError
 * @throws RuntimeError
 * @throws SyntaxError
 * @throws \Exception
 */
public function display(): void
{
    $this->twig->display($this->templatePath, [
        'page' => $this->currentPage,
        'pages' => $this->getPages(),
        'route' => $this->route
    ]);
}
```

Nous allons ensuite ajouter à notre fichier config/services.yaml l'argument du templatePath :

```yaml
# add more service definitions when explicit configuration is needed
# please note that last definitions always *replace* previous ones
App\Services\Pagination:
    arguments:
        $templatePath: '/partials/pagination.html.twig'
```

Et maintenant créer le template de pagination correspondant à ce chemin :

```twig
<div class="col-12 center">
    <ul class="pagination">
        <li class="page-item {% if page == 1 %}disabled{% endif %}">
            <a class="page-link" href="{{ path(route, {'page': page - 1}) }}">&laquo;</a>
        </li>
        {% for i in 1..pages %}
            <li class="page-item {% if page == i %}active{% endif %}">
                <a class="page-link" href="{{ path(route, {'page': i}) }}">{{ i }}</a>
            </li>
        {% endfor %}
        <li class="page-item {% if page == pages %}disabled{% endif %}">
            <a class="page-link" href="{{ path(route, {'page': page + 1}) }}">&raquo;</a>
        </li>
    </ul>
</div>
```

Nous devons ensuite modifier notre fonction index du BlogController afin d'utiliser notre service et lui donner les informations nécessaires à son fonctionnement, il faut noter aussi l'ajout d'un paramètre à la route qui représente le numéro de page et qui par défaut vaut 1.

Nous remplaçons les données envoyées dans la variable articles par nos données paginées.

```php
/**
 * Permet d'afficher la liste des thèmes et des articles
 * @param Pagination $pagination
 * @param $page
 * @return Response
 * @throws Exception
 */
#[Route('/blog/{page<\d+>?1}', name: 'blog', requirements: ["page" => "\d+"])]
public function index(Pagination $pagination, $page): Response
{
    $articles = $this->entityManager->getRepository(Articles::class)->findAll();
    $themes = $this->entityManager->getRepository(Themes::class)->findAll();

    $pagination
        ->setOrderBy(['createdAt' => 'DESC'])
        ->setPage($page)
        ->setEntityClass( entityClass: Articles::class);

    $paginateRessources = $pagination->getData();

    return $this->render( view: 'blog/index.html.twig', [
        'articles' => $paginateRessources,
        'themes' => $themes,
        'current_menu' => 'blog',
        'pages' => $pagination->getPages(),
        'page' => $page,
        'pagination' => $pagination,
    ]);
}
```

Et enfin modifier notre template pour afficher notre pagination :

```twig
<!-- blog section start -->
<div class="container-md mt-5">
    <div class="container">
        <div class="row">
            {% for article in articles|sort((a, b) => b.createdAt <=> a.createdAt) %}
                {% if article.isPrivate == 0 %}
                    {% include '/partials/single_article.html.twig' %}
                {% else %}
                    {% if app.user %}
                        {% include '/partials/single_article.html.twig' %}
                    {% else %}
                        {% include '/partials/no-access.html.twig' %}
                    {% endif %}
                {% endif %}
            {% endfor %}
        </div>
    </div>
</div>
<!-- blog section  end -->
<div class="flex justify-center items-center">
    {% include './partials/pagination.html.twig' with {'route': 'blog'} %}
</div>
```

Nous avons maintenant une pagination fonctionnelle, elle peut être améliorer et même s'étoffer, on peut imaginer créer des fonctions au sein de notre service en fonction de nos besoins. Nous pourrions aussi ajouter un affichage conditionnel pour le cas où le blog comporte moins de 5 articles ce qui semble peu probable.

TP6 BARRE DE FILTRE

Vous allez devoir mettre en place une barre de recherche sur nos articles que ce soit sur le champ title ou content.

Le but étant de permettre à l'utilisateur de filtrer les articles en fonction d'une chaîne de caractère. Vous pourrez utiliser une simple classe en ajoutant un dossier Classe à votre src.

Vous rédigerez une méthode au sein du repository des articles afin d'effectuer une requête prenant en compte la string de la recherche.

Il faudra donc bien évidemment créer un formulaire.

TP7 CRÉER UN SYSTÈME DE COMMENTAIRES

A ce stade vous avez toutes les cartes en main afin de créer une nouvelle fonctionnalité au sein de notre blog.

Vous allez devoir ajouter un système de commentaires, il faudra que l'utilisateur ayant rédigé le commentaire puisse le modifier. L'administrateur doit pouvoir modérer les commentaires rapidement.

Vous êtes libre de la mise en place, ne laisser la possibilité que de ne poster un commentaire par article ou plusieurs. L'affichage de ceux-ci est libre également.