



SYMFONY

Un mémo



BASES

Commandes et débuts



```
symfony new <project_name>
# Initialiser un project Symfony

symfony check:req
# Vérifie que que votre systeme
# peut faire tourner le projet

symfony serve
# Pour lancer un serveur local
# Ctrl + C pour le kill

symfony serve -d
# Lancer un serveur en background

symfony serve:status
# Pour connaitre l'état du serveur

symfony serve:stop
# Pour arrêter le serveur
```

Les commandes de base

- Voici les commandes de base pour créer votre projet et manipuler le serveur local

La console Symfony

- La console Symfony vous donne accès à tous les utilitaires de votre projet, elle vient avec deux commandes quasiment identiques

```
php ./bin/console
symfony console
# Affiche toutes les options disponibles
# via la console

# Les deux commandes sont identiques à une
# différence près : 'symfony console'
# injecte des variables d'environnement
# propres au projet, il est donc préférable
# d'utiliser cette commande
```

Quelques commandes de base

```
symfony console debug:autowiring <recherche>  
# Rechercher dans les services en autowiring
```

```
symfony console debug:container <recherche>  
# Permet de rechercher n'importe quel service
```

```
symfony console debug:container --parameters  
# Affiche la liste des paramètres de Symfony
```

```
symfony console cache:clear  
# Purge le cache
```

- Voici quelques commandes que vous allez probablement utiliser tous les jours

Les bundle de base

- Voici quelques bundles avec lesquels on a commencé notre projet

```
composer require annotations  
# Installe sensio/framework-extra-bundle
```

```
composer require templates  
# Installe symfony/twig-pack
```

```
composer require debug  
# Installe symfony/debug-pack  
# Qui contient :  
#   - symfony/debug-bundle  
#   - symfony/monolog-bundle  
#   - symfony/profiler-pack
```

```
composer require asset  
# Installe symfony/asset
```

Des bundles avancés

- Voici quelques autres bundles plus pointus

```
composer require orm
```

```
# Installe le "orm-pack" avec Doctrine
```

```
composer require maker --dev
```

```
# Installe le maker bundle seulement en DEV
```

```
composer require knplabs/knp-time-bundle
```

```
# Un bundle pour rendre les dates relatives
```

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

```
# Un bundle pour créer des fixtures dans Doctrine
```

```
composer require zenstruck/foundry --dev
```

```
# Un bundle pour créer des fixtures plus intéressantes
```

```
# avec faker
```

```
composer require stof/doctrine-extensions-bundle
```

```
# Un bundle d'extensions Doctrine pour ajouter
```

```
# sluggable, timestampable...
```

```
# Attention, il faut les activer manuellement
```

```
composer require knplabs/knp-paginator-bundle
```

```
# Pour installer le knp-paginator-bundle
```

Bundles Expert

- Des bundles pour des tâches plus poussées

```
composer require security  
# Installe symfony/security-bundle
```

```
composer require serializer  
# Installe le serializer de Symfony
```

```
composer require form  
# Pour installer le gestionnaire de  
# formulaires Symfony
```

```
composer require validator  
# Le système de validation pour  
# les formulaires de Symfony
```

```
composer require symfony/mime  
# L'extension de gestion de mime  
# pour Symfony
```

```
composer require liip/imaginary-bundle  
# Bundle pour manipuler les images  
# y compris créer des thumbnails
```

```
composer require league/flysystem-bundle  
# Un bundle pour mieux gérer nos fichiers  
# localement ou sur le cloud
```


Services.yaml | services_dev.yaml

```
parameters:  
  # Je pourrai appeler le paramètre '%cache_system%'  
  # depuis n'importe quel autre fichier de config  
  # ou controller et il vaudra 'cache.adapter.filesystem'  
  cache_system: cache.adapter.filesystem
```

- Ce sont les deux facettes du même fichier, un général et un spécifique au dev
- Je peux créer de nouveaux paramètres dedans

Exemples

- Voici deux exemples, l'un dans un fichier de config, l'autre depuis un controller

```
framework:  
    cache:  
        app: '%cache_system%'
```

```
/**  
 * @Route("/", name="app_homepage")  
 * @return Response  
 */  
public function homepage(QuestionRepository $repository)  
{  
    $cache_system = $this->getParameter( name: 'cache_system');
```

Injecter manuellement

- Il est possible de rendre n'importe quel paramètre ou service « autowireable » dans service.yaml

```
services:
  _defaults:
    autowire: true
    autoconfigure: true
    bind:
      # Tout ça sera disponible en autowiring depuis n'importe quel service

      # Si j'injecte une variable bool $isDebug,
      # j'aurai la valeur de paramètre %kernel.debug%
      bool $isDebug: '%kernel.debug%'

      # Si j'injecte un service avec le type-hint
      # Psr\Log\LoggerInterface ET le nom de variable $mdLogger
      # Je vais avoir le service avec l'ID monolog.logger.markdown
      Psr\Log\LoggerInterface $mdLogger: '@monolog.logger.markdown'

      # Et ainsi de suite
      string $appSecret: '%kernel.secret%'
      string $projectDir : '%kernel.project_dir%'
      string $publicPath : '%kernel.project_dir%/public'
```

Exemple

- Voici un exemple de deux injections faites grâce à cette technique

```
public function __construct(MarkdownParserInterface $markdownParser,  
                           CacheInterface           $cache,  
                           bool                     $isDebug,  
                           LoggerInterface          $mdLogger,  
                           Security                 $security)  
{
```

Variables d'environnement

- En plus des variables d'environnement classiques, Symfony met à notre disposition deux fichiers
 - « *.env* » : *qui est commit sur git*
 - « *.env.local* » : *qui n'est pas commit sur git*
- La valeur de ces variables sera écrasée dans l'ordre suivant :
 - « *.env* » sera écrasé par « *.env.local* »
 - « *.env.local* » sera écrasé par les variables d'environnement classiques
- Les variables d'environnement sont lues comme des paramètres

```
sentry:  
    dsn: '%env(SENTRY_DSN)%'
```

Secret Vault

```
symfony console secret:set <nom_du_secret>
# Pour enregistrer une nouvelle valeur
# dans le coffre secret de DEV

symfony console secret:set <nom_du_secret> --env=prod
# Pour enregistrer une nouvelle valeur
# dans le coffre secret de PROD

symfony console secret:list
# Pour récupérer la liste des valeurs dans
# notre coffre actuel

symfony console secret:list --reveal
# Pour afficher ces valeurs
# si on possède la clé privée

symfony console secret:list --env=prod --reveal
# Pour les afficher dans un environnement
# particulier
```

- On peut créer plusieurs Vault : Prod et Dev
- Par défaut, la clé privée Dev est commit sur git, pas la clé privée prod
- Les variables d'environnement écrasent les valeurs du Vault

Maker Bundle

- Un des outils les plus pratiques de Symfony
- Il n'est utilisé qu'en dev

```
composer require maker --dev  
# Installe le maker bundle seulement en DEV  
# Nous n'en aurons pas besoin en prod, c'est un détail
```

```
make
make:auth
make:command
make:controller
make:crud
make:docker:database
make:entity
make:factory
make:fixtures
make:form
make:message
make:messenger-middleware
make:migration
make:registration-form
make:reset-password
make:serializer:encoder
make:serializer:normalizer
make:story
make:subscriber
make:test
make:twig-extension
make:user
make:validator
make:voter
```

Liste de ses commandes

- Il peut faire beaucoup de choses pour nous et la liste grandit au fur et à mesure que vous installez des bundles

DOCTRINE

Et les entités



Doctrine

- Les commandes pour installer Doctrine et créer une BDD Docker

```
composer require orm  
# Pour installer le "orm-pack"  
# Avec Doctrine et plein de choses
```

```
./bin/console make:docker:database  
# Faire un docker-compose.yaml automatiquement
```

```
docker-compose up -d  
# Lance mon container
```

```
docker-compose down  
# Stop et détruit mes containers
```

```
docker-compose ps  
# Montre les process en cours
```

Workflow

Entity (create or update)



make:migration



doctrine:migrations:migrate

- Une entité est une classe comme une autre, Doctrine sait quoi faire avec grâce aux annotations
- On doit penser Objet et pas SQL

Ecrire / Update en DB

- On utilise le service « EntityManagerInterface »
- On crée notre objet
- On demande à Doctrine de l'ajouter à la liste des choses à persister
- On flush pour inscrire tout ce qui était sur la liste

```
public function create(EntityManagerInterface $entityManager, Request $request): Response
{
    $user = new User();
    $user->setFirstName($request->request->get( key: 'firstName'))
        ->setLastName($request->request->get( key: 'lastName'))
        ->setEmail($request->request->get( key: 'email'));

    $entityManager->persist($user);
    $entityManager->flush();
}
```

Lire en DB avec le repository

- On injecte le repository de l'entité correspondante et on appelle la méthode voulue

```
/**
 * @Route("/", name="app_homepage")
 * @return Response
 */
public function homepage(QuestionRepository $repository)
{
    $questions = $repository->findAllAskedOrderByNewest();

    return $this->render( view: 'question/homepage', [
        'questions' => $questions
    ]);
}
```

Lire en DB avec le Param Converter

```
/**
 * @param User $user
 * @return Response
 * @Route("/user/{email}", name="app_user_show")
 * @IsGranted("USER_VIEW", subject="user")
 */
public function show(User $user): Response
{
    return $this->render(view: 'user/user_show', [
        'user' => $user
    ]);
}
```

- Doctrine va lire les paramètres d'URL et voir si un attribut a le même nom et aller le chercher automatiquement

DQL

- C'est le langage de query utilisé par Doctrine dans les repositories

```
/**
 * @return Question[] Returns an array of Question objects
 */
public function findAllAskedOrderByNewest()
{
    return $this->createQueryBuilder( alias: 'q')
        ->andWhere('q.askedAt IS NOT NULL')
        ->leftJoin( join: 'q.tag', alias: 'tag')
        ->addSelect( select: 'tag')
        ->leftJoin( join: 'q.user', alias: 'user')
        ->addSelect( select: 'user')
        ->orderBy( sort: 'q.askedAt', order: 'DESC')
        ->getQuery() // Pour effectivement créer la query
        ->getResult(); // Récupère les résultats
}
```

Criteria

- Objet permettant de modifier une query avant qu'elle ne soit faite
- Très utile dans une entité pour filtrer les résultats

```
/**
 * @return Criteria
 */
public static function createApprovedCriteria(): Criteria
{
    return Criteria::create()
        ->andWhere(Criteria::expr()->eq('status', Answer::APPROVED));
}
```

```
/**
 * @return Collection
 */
public function getApprovedAnswers(): Collection
{
    return $this->answers->matching(AnswerRepository::createApprovedCriteria());
}
```



```
public function new(Request $request): Response
{
    // Tous les paramètres $_GET
    $request->query->all();

    // Tous les $_POST
    $request->request->all();

    // Tous les fichiers
    $request->files->all();

    // Interroge la méthode
    $request->isMethod( method: 'POST');
```

L'objet Request

- Il est disponible en autowire

Les objets Response

- Il en existe de plusieurs types selon ce que l'on veut faire
- Beaucoup sont disponibles avec des raccourcis de « AbstractController »

```
// Un render via Twig en y passant des paramètres
return $this->render( view: 'question/homepage', [
    'questions' => $questions
]);
```

```
// Une redirection avec des paramètres à passer en URL
return $this->redirectToRoute( route: '/questions/{slug}', [
    'slug' => $question->getSlug()
]);
```

```
// Une réponse encodée en JSON avec le serializer
// s'il est installé en précisant les groupes à encoder
return $this->json($questions, status: 200, [], [
    'groups' => ['main']
]);
```

Les objets Response

```
// La réponse la plus classique en passant une string  
return new Response( content: 'yeah gagné');
```

- Il est aussi possible de les instancier directement

```
$response = new StreamedResponse(function () {  
    // TODO - Comportement du stream  
});
```

SÉCURITÉ

Mémo utiles



Nouveau composant de sécurité

- Depuis Symfony 5.1, on a un nouveau composant de sécurité utilisant les Passeports qui remplace le système Guard, on doit préciser son activation
- Toutes les configurations sont dans « security.yaml »

```
security:  
    # https://symfony.com/doc/current/security/authenticator\_manager.html  
    enable_authenticator_manager: true
```

Classe User

- Pour être utilisée dans le système de sécurité, une classe User doit implémenter « `UserInterface` »
- Si elle utilise un Password qui est géré depuis notre app, elle doit aussi implémenter « `PasswordAuthenticatedUserInterface` »

```
class User implements UserInterface, PasswordAuthenticatedUserInterface  
{
```

Déclarer l'User_provider

```
providers:
  app_user_provider:
    entity:
      class: App\Entity\User
      property: email
```

- On doit déclarer dans security.yaml quelle sera la classe à utiliser pour l'authentification et avec quel attribut

Déclarer l'encodage

- Il faut aussi déclarer quel algorithme est utilisé pour l'encodage des passwords
- La liste est lue de haut en bas et s'arrête dès qu'elle trouve un match

```
password_hashers:  
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'  
    App\Entity\User:  
        algorithm: auto
```


Encoder un password

- Le service « UserPasswordHasherInterface » pour encoder nos passwords
- L'algorithme est choisi en fonction de l'entité que l'on passe en premier argument

```
$user->setPassword($this->hasher->hashPassword($user, plainPassword: 'password'));
```

```
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    lazy: true
    provider: app_user_provider
    custom_authenticator:
      - App\Security\LoginFormAuthenticator
      - App\Security\ApiTokenAuthenticator
    logout:
      path: app_logout
      # where to redirect after logout
      # target: app_any_route
    remember_me:
      secret: '%kernel.secret%'
      lifetime: 604800
    switch_user: true
```

Firewall

- Systèmes chargés de l'authentification
- Définit les authenticators utilisés et pour quels chemins
- On peut déclarer autant de firewall que l'on veut

Authenticator

```
interface AuthenticatorInterface
{
    /** Does the authenticator support the given Request? ...*/
    public function supports(Request $request): ?bool;

    /** Create a passport for the current request. ...*/
    public function authenticate(Request $request): PassportInterface;

    /** Create an authenticated token for the given user. ...*/
    public function createAuthenticatedToken(PassportInterface $passport, string $firewallName): TokenInterface;

    /** Called when authentication executed and was successful! ...*/
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response;

    /** Called when authentication executed, but failed (e.g. wrong username password). ...*/
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception): ?Response;
}
```

- Doit implémenter
« AuthenticatorInterface » et donc
ces méthodes :
 - *Supports* : Quand est
utilisé cet authenticator
 - *Authenticate* : Crée un
passeport
 - *createAuthenticatedToken* :
Attribut un token
d'identification
 - *onAuthenticationSuccess* :
Quoi faire si ça marche
 - *onAuthenticationFailure* :
quoi faire si ça merde

Passeports avec password

- Le passeport standard 3 arguments et n'est valide que si tout marche :
 - *Le badge utilisateur ou un callback*
 - *Le Password ou un callback*
 - *Des options*
- Le UserBadge réagit en fonction du UserProvider
- Le password en fonction de l'algorithme passé

```
public function authenticate(Request $request): PassportInterface
{
    $email = $request->request->get( key: 'email', default: '');

    return new Passport(
        new UserBadge($email),
        new PasswordCredentials($request->request->get( key: 'password', default: '')),
        [
            new CsrfTokenBadge( csrfTokenId: 'authenticate', $request->request->get( key: '_csrf_token' )),
            new RememberMeBadge()
        ]
    );
}
```

SelfValidatingPassport

```
public function authenticate(Request $request): PassportInterface
{
    $token = str_replace( search: 'Bearer ', replace: '', $request->headers->get( key: 'authorization' ));
    return new SelfValidatingPassport(new UserBadge($token, function ($token) {
        $tokenObj = $this->tokenRepository->findOneBy(['token' => $token]);
        if (!$tokenObj) {
            throw new CustomUserMessageAuthenticationException( message: 'Invalid Token');
        }
        if ($tokenObj->isExpired()) {
            throw new CustomUserMessageAuthenticationException( message: 'Expired Token');
        }
        return $tokenObj->getUser();
    }));
}
```

- Ces passeports n'ont pas besoin de password et sont valides s'il trouve un User et que les options se résolvent (CSRF par exemple)

Authentifier dans un controller

```
return $authenticator->authenticateUser(  
    $newUser,  
    $loginFormAuthenticator,  
    $request  
);
```

```
public function new(Request $request,  
    EntityManagerInterface $entityManager,  
    UserPasswordHasherInterface $hasher,  
    UserAuthenticatorInterface $authenticator,  
    LoginFormAuthenticator $loginFormAuthenticator): Response  
{
```

- Pour authentifier dans un controller, j'utilise le service « UserAuthenticatorInterface » qui va prendre 3 arguments :
 - *L'utilisateur à authentifier*
 - *L'authenticator à utiliser (il faut l'autowire)*
 - *La requête*
- Ce service répond avec les méthodes de succès ou d'échec de l'authenticator

Protéger des zones

- On utilise access_control de security.yaml pour protéger des zones

```
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    # IS_AUTHENTICATED_ANONYMOUSLY : tous les utilisateurs ont ce droit
    # authentifié ou non
    - { path: ^/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
    # A partir du moment ou je suis authentifié, j'ai ce droit
    # quelque soit mon rôle (c'est équivalent à check ROLE_USER
    # puisque mon entité donne ce rôle à tout le monde)
    - { path: ^/, role: IS_AUTHENTICATED_FULLY }
```


Protéger avec les annotations

- Je protège des routes particulières avec des annotations dans mon controller

```
/**  
 * @Route("/questions/new", name="app_question_new")  
 * @IsGranted("ROLE_USER")  
 */  
public function new(Request $request, EntityManagerInterface $em, UploadHelper $helper): Response  
{
```


Protéger en PHP

- Je peux aussi protéger mes routes en PHP quand j'ai besoin de passer un argument particulier que l'annotation ne peut pas avoir

```
public function downloadReference(QuestionReference $reference, UploadHelper $helper)
{
    $question = $reference->getQuestion();
    $this->denyAccessUnlessGranted( attribute: 'EDIT_QUESTION', $question);
}
```

Hiérarchie de rôle

- Je déclare des pseudo rôles qui englobent d'autres
- Je peux utiliser ces rôles comme des rôles normaux

```
role_hierarchy:
```

```
  ROLE_SUPER_ADMIN: [ROLE_ADMIN_USER, ROLE_ADMIN_QUESTION, ROLE_ADMIN_ANSWER, ROLE_ALLOWED_TO_SWITCH]
```

```
  ROLE_EDITOR: [ROLE_ADMIN_QUESTION, ROLE_ADMIN_ANSWER]
```

Les voters

- Ce sont les entités responsables de l'autorisation à une zone, elles vont lire les rôles de l'utilisateur et, si le rôle est dans leur champs d'actions, vont voter pour ou contre l'accès
- Par défaut, on à 2 voters :
 - *RoleVoter* (pour tout ce qui commence par « *ROLE_* »)
 - *AuthenticatedVoter* (pour les mots clés du style « *IS_ANONYMOUS* »)
- Un Voter doit hériter de « Voter » qui implémente « VoterInterface »

```
abstract class Voter implements VoterInterface
{
    /** {@inheritdoc} ...*/
    public function vote(TokenInterface $token, $subject, array $attributes){...}

    /** Determines if the attribute and subject are supported by this voter. ...*/
    abstract protected function supports(string $attribute, $subject);

    /** Perform a single access check operation on a given attribute, subject and token. ...*/
    abstract protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token);
}
```

Les méthodes

- Support définit quand le voter est compétent, s'il n'est pas compétent il s'abstient
- voteOnAttribute est la logique de vote
- Le voter retourne un booléen

```
class QuestionVoter extends Voter
{
    protected function supports(string $attribute, $subject): bool
    {
        return in_array($attribute, ['POST_EDIT', 'POST_VIEW'])
            && $subject instanceof \App\Entity\Question;
    }
}
```

voteOnAttribute

```
protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
{
    $user = $token->getUser();
    // if the user is anonymous, do not grant access
    if (!$user instanceof UserInterface) {
        return false;
    }

    // ... (check conditions and return true to grant permission) ...
    switch ($attribute) {
        case 'POST_EDIT':
            // logic to determine if the user can EDIT
            // return true or false
            break;
        case 'POST_VIEW':
            // logic to determine if the user can VIEW
            // return true or false
            break;
    }

    return false;
}
```

- Je n'ai pas besoin de déclarer mes Voters ou mes nouveaux rôles

Passer le sujet

- Je dois par contre passer le sujet (l'entité) si je cherche à identifier via un Voter Custom

```
/**
 * @param User $user
 * @return Response
 * @Route("/user/{email}", name="app_user_show")
 * @IsGranted("USER_VIEW", subject="user")
 */
public function show(User $user): Response
{
    return $this->render('user/user_show', [
        'user' => $user
    ]);
}
```

SYMPHONY FROMS

Le système complet



FormType

```
abstract class AbstractType implements FormTypeInterface
{
    /** {@inheritDoc} ...*/
    public function buildForm(FormBuilderInterface $builder, array $options){...}

    /** {@inheritDoc} ...*/
    public function buildView(FormView $view, FormInterface $form, array $options){...}

    /** {@inheritDoc} ...*/
    public function finishView(FormView $view, FormInterface $form, array $options){...}

    /** {@inheritDoc} ...*/
    public function configureOptions(OptionsResolver $resolver){...}

    /** {@inheritDoc} ...*/
    public function getBlockPrefix(){...}

    /** {@inheritDoc} ...*/
    public function getParent(){...}
}
```

- Une entité de formulaire est un Type et hérite de « AbstractType » qui implémente « FormTypeInterface »

buildForm()

- C'est la méthode pour construire les champs, on va y passer leur slug, leur Type (text, textarea, file...) et les options que l'on veut passer au Type

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add( child: 'email', type: EmailType::class)
        ->add( child: 'firstName')
        ->add( child: 'lastName')
        ->add( child: 'plainPassword', type: PasswordType::class, [
            'mapped' => false,
            'constraints' => [
                new NotBlank(),
                new Length([
                    'min' => 5,
                    'minMessage' => 'Un plus long password steuplé...'
                ])
            ]
        ])
    }
}
```

Lier une classe

- Un lien en classe dans les options par défaut de notre Type
- Ces options sont passées sous la forme d'un array \$option dans les autres méthodes, je peux aussi en passer quand je crée le formulaire dans mon controller

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

Render

```
/**
 * @Route("/questions/new", name="app_question_new")
 */
public function new(): Response
{
    // Pour créer l'objet formulaire
    $form = $this->createForm( type: QuestionFormType::class);

    return $this->render( view: 'question/new.html.twig', [
        // Je passe la vue créée, pas l'objet
        'questionForm' => $form->createView()
    ]);
}
```

- On procède en deux étapes :
 - On crée le formulaire dans le contrôleur
 - Puis on passe la vue dans le template Twig
- Je peux passer des données et des options quand je crée le formulaire

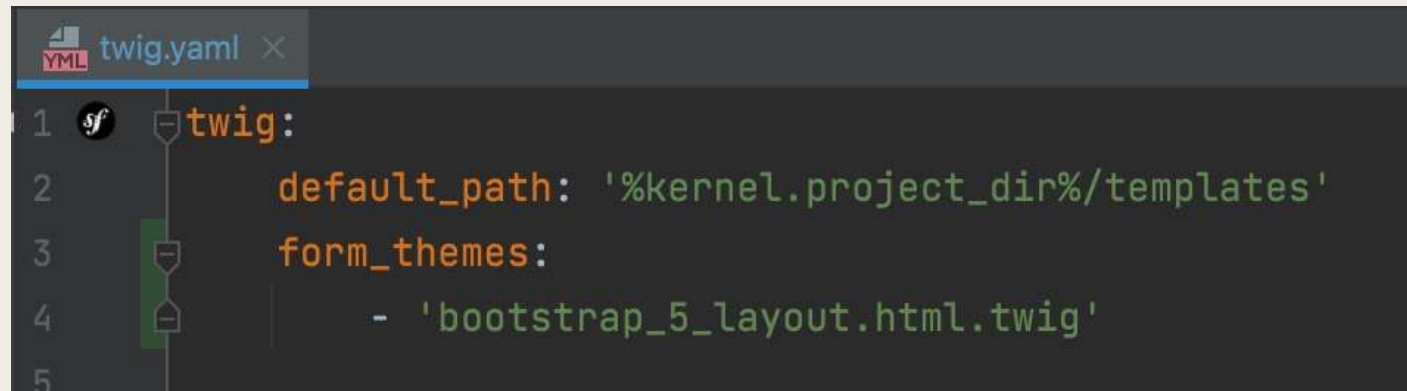
Twig

- On peut render le formulaire d'un coup ou par petits bouts dans twig
- Je peux passer des options à chacun des champs

```
{{ form_start(questionForm) }}  
{{ form_row(questionForm.name) }}  
{{ form_row(questionForm.question) }}  
{{ form_row(questionForm.imageFile) }}  
{{ form_row(questionForm.askedAt) }}  
{{ form_row(questionForm.user) }}  
{{ form_row(questionForm.submit) }}  
{{ form_end(questionForm) }}
```

Thèmes

- Il existe des thèmes par défaut pour les formulaires, on les configure dans Twig.yaml



```
YML twig.yaml x
1 sf twig:
2   default_path: '%kernel.project_dir%/templates'
3   form_themes:
4     - 'bootstrap_5_layout.html.twig'
5
```

The screenshot shows a code editor with a file named 'twig.yaml'. The content of the file is a YAML configuration for Twig. It defines a 'twig' section with a 'default_path' pointing to '%kernel.project_dir%/templates' and a 'form_themes' list containing 'bootstrap_5_layout.html.twig'. The editor has a dark theme and a sidebar on the left showing a file tree with a folder icon and a file icon.

Gérer le submit

```
/**
 * @Route("/questions/new", name="app_question_new")
 */
public function new(Request $request): Response
{
    $form = $this->createForm( type: QuestionFormType::class);

    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        dd($form->getData());
    }

    return $this->render( view: 'question/new.html.twig', [
        'questionForm' => $form->createView()
    ]);
}
```

- Je gère le submit dans mon controller en récupérant les données déjà process par le formulaire
- Si une classe est liée, je vais recevoir une entité avec les attributs remplis par les champs du formulaire

Champs non mappés

- Si un champs ne correspond pas à un attribut de notre entité (à un getter et un setter pour être précis), il est possible de ne pas le mapper
- Et de le récupérer dans le controller pour en gérer manuellement le traitement

```
$form = $this->createForm( type: RegistrationFormType::class);

$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) {
    /** @var $newUser User */
    $newUser = $form->getData();

    $plainPassword = $form['plainPassword']->getData();
    $newUser->setPassword($hasher->hashPassword($newUser, $plainPassword));

    $entityManager->persist($newUser);
    $entityManager->flush();
}
```

Validation : @Assert

- On peut spécifier des vérifications spécifiques coté serveur directement sur notre entité si le champs est mappé avec l'entité

```
/**
 * @ORM\Column(type="string", length=180, unique=true)
 * @Groups("main")
 * @Assert\NotBlank()
 * @Assert\Email()
 */
private $email;
```


@Assert | Callback()

```
/**
 * @Assert\Callback()
 */
public function validate(ExecutionContextInterface $context)
{
    if (strpos($this->getName(), 'Francis') !== false) {
        $context->buildViolation('On avait dit pas Francis !')
            ->atPath('name')
            ->addViolation();
    }
}
```

- Pour créer une validation perso dans son entité
- buildViolation : crée la violation et passe le message
- atPath : l'attribut sur lequel on le spécifie
- addViolation : pour enregistrer

Contraintes et champs non mappés

- Je peux ajouter l'option « constraints » pour rajouter des contraintes à des champs non mappés

```
$builder
->add( child: 'email', type: EmailType::class)
->add( child: 'firstName')
->add( child: 'lastName')

->add( child: 'plainPassword', type: PasswordType::class, [
  'mapped' => false,
  'constraints' => [
    new NotBlank(),
    new Length([
      'min' => 5,
      'minMessage' => 'Un plus long password steuplé...'
    ])
  ]
])

->add( child: 'submit', type: SubmitType::class);
```

Validator hors formulaire

```
/** @var UploadedFile $file */
$file = $request->files->get( key: 'reference');

$violations = $validator->validate($file, [
    new NotBlank(),
    new File([
        'mimeType' => [
            'image/*',
            'application/pdf',
        ]
    ])
]);

if ($violations->count() > 0) {
    $this->addFlash( type: 'error', $violations[0]->getMessage());
} else {
```

- Je peux utiliser le validator en dehors d'un formulaire avec le service « ValidatorInterface »

Types de champs custom

- Les Types de champs sont des Types exactement comme les FormType
- Pour partir d'une base, on les fait hériter avec la méthode « getParent »

```
class UserSelectTextType extends AbstractType
{
    // Les types n'utilisent pas un vrai système
    // d'héritage mais cette fonction signifie que
    // à moins qu'on le spécifie autrement, cette classe
    // se comportera comme un EmailType
    public function getParent()
    {
        return EmailType::class;
    }
}
```

DataTransformer

```
class UserSelectTextType extends AbstractType
{
    private UserRepository $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // Je le ré-injecte ici
        $builder->addModelTransformer(new EmailToUserTransformer($this->userRepository));
    }
}
```

- Créer un nouveau Type de champs permet de créer une logique de transformation
- Les DataTransformer ne sont pas considérés comme des services, il faut faire l'injection de dépendance manuellement depuis le Type

DataTransformer

- Ils doivent implémenter « DataTransformerInterface »
 - *transform* : de l'entité vers le formulaire
 - *reverseTransform* : du formulaire vers l'entité

```
interface DataTransformerInterface
{
    /** Transforms a value from the original representation to a transformed representation. ...*/
    public function transform($value);

    /** Transforms a value from the transformed representation to its original ...*/
    public function reverseTransform($value);
}
```

Extensions

- Elles permettent de rajouter du code à un Type existant
- Elles doivent hériter de AbstractTypeExtension qui implémente FormTypeExtensionInterface
- Les méthodes sont les mêmes que pour les Types mais sera exécuté après pour étendre ou réécrire le Type de base

```
abstract class AbstractTypeExtension implements FormTypeExtensionInterface
{
    /** {@inheritDoc} ...*/
    public function buildForm(FormBuilderInterface $builder, array $options){...}

    /** {@inheritDoc} ...*/
    public function buildView(FormView $view, FormInterface $form, array $options){...}

    /** {@inheritDoc} ...*/
    public function finishView(FormView $view, FormInterface $form, array $options){...}

    /** {@inheritDoc} ...*/
    public function configureOptions(OptionsResolver $resolver){...}
}
```

Configuration

- Pour préciser quelle Type on étend, il faut utiliser la méthode « getExtendedTypes »

```
class TextareaSizeExtension extends AbstractTypeExtension
{
    public static function getExtendedTypes(): iterable
    {
        return [TextareaType::class];
    }
}
```


UPLOADS

Les fichiers



UploadedFile

```
/**
 * @Route("/questions/test/upload", name="app_test_upload")
 */
public function uploadTest(Request $request)
{
    /** @var UploadedFile $newFile */
    $newFile = $request->files->get( key: 'image');
    $destination = $this->getParameter( name: 'kernel.project_dir') . '/public/uploads';

    // C'est déjà mieux
    $fileName = uniqid() . '-' . $newFile->getClientOriginalName();

    $newFile->move($destination, $fileName);

    return new Response( content: 'yeah gagné');
}
```

- Symfony me passe les fichiers sous forme d'objet « UploadedFile » avec des méthodes pour le manipuler
- Cette classe hérite de la classe générique « File »

Un Helper

- On va généralement créer un service Helper pour centraliser la logique et éviter de saturer le code du controller

```
class UploadHelper
{
    private string $projectDir;

    public function __construct(string $projectDir)
    {
        $this->projectDir = $projectDir;
    }

    public function uploadQuestionImage(UploadedFile $file): string
    {
        $destination = $this->projectDir . '/public/uploads/questions_images';
        $originalFileName = $file->getClientOriginalName();
        $baseFileName = pathinfo($originalFileName, flags: PATHINFO_FILENAME);
        $fileName = Urlizer::urlize($baseFileName) . '-' . uniqid() . '.' . $file->guessExtension();
        $file->move($destination, $fileName);

        return $fileName;
    }
}
```

Centraliser l'info

```
class UploadHelper
{
    const QUESTION_IMAGE = 'uploads/questions_images';
    const DEFAULT_IMAGE = 'images/cute_cat.jpg';

    private string $publicPath;

    public function __construct(string $publicPath)
    {
        $this->publicPath = $publicPath;
    }

    /**
     * @param UploadedFile $file
     * @return string
     */
    public function uploadQuestionImage(UploadedFile $file): string
    {
        $destination = $this->publicPath . '/' . self::QUESTION_IMAGE;
        $originalFileName = $file->getClientOriginalName();
```

- On va centraliser dans le Helper avec des constantes les chemins d'accès pour ne pas avoir à fouiller partout en cas de modification
- On va utiliser ces constantes dans nos entités

Filesystem

- C'est l'objet qui nous permet de manipuler des fichiers facilement

```
public function fixtureUpload(File $file): string
{
    $destination = $this->publicPath . '/' . self::QUESTION_IMAGE;
    $originalFileName = $file->getFilename();
    $baseFileName = pathinfo($originalFileName, flags: PATHINFO_FILENAME);
    $fileName = Urlizer::urlize($baseFileName) . '-' . uniqid() . '.' . $file->guessExtension();

    $fs = new Filesystem();
    $fs->copy($file->getRealPath(), targetFile: $destination . '/' . $fileName, overwriteNewerFiles: true);

    return $fileName;
}
```

Des entités

- On peut avoir besoin de créer des entités pour des fichiers et ainsi utiliser les relations Doctrine pour lier des fichiers à d'autres entités

```
class QuestionReference
{
    /** @ORM\Id ...*/
    private $id;

    /** @ORM\ManyToOne(targetEntity=Question::class, inversedBy="questionReferences") ...*/
    private $question;

    /** @ORM\Column(type="string", length=255) ...*/
    private $filename;

    /** @ORM\Column(type="string", length=255) ...*/
    private $originalFilename;

    /** @ORM\Column(type="string", length=255) ...*/
    private $mimeType;
```

Les stream

```
// Crée un stream en lecture pour le fichier
// que le client vient d'uploader
$stream = fopen($file->getPathname(), mode: 'r');

// Crée un nouveau fichier et ouvre un stream en écriture
$destination = fopen(filename: self::QUESTION_IMAGE . '/' . $fileName, mode: 'wb');

// Copie un le stream de l'un vers l'autre
stream_copy_to_stream($stream, $destination);

// On ferme tout
fclose($stream);
fclose($destination);
```

- Pour éviter de surcharger la mémoire avec des fichiers trop gros, il est conseillé d'utiliser des streams, [plus d'infos ici](#)

Afficher un stream

- Symfony à un objet pour ça mais il faut lui spécifier son comportement
- Pour que le navigateur comprenne ce qu'il se passe, il faut aussi lui passer les bons headers

```
public function downloadReference(QuestionReference $reference, UploadHelper $helper)
{
    $response = new StreamedResponse();
    $response->setCallback(function () use ($reference, $helper) {
        $outputStream = fopen('php://output', 'wb');
        $fileStream = $helper->readPrivateStream($reference->getFilePath());
        stream_copy_to_stream($fileStream, $outputStream);
    });

    $response->headers->set('Content-Type', $reference->getMimeType());

    $disposition = HeaderUtils::makeDisposition(
        disposition: HeaderUtils::DISPOSITION_INLINE,
        Urlizer::urlize($reference->getOriginalFilename())
    );
    $response->headers->set('Content-Disposition', $disposition);

    return $response;
}
```