DOCUMENTATION SUR SYMFONY (FRAMEWORK PHP)

I. NOTION DE FRAMEWORK

Un **framework** est, comme son nom l'indique en anglais, un "cadre de travail". L'objectif d'un framework est généralement de simplifier le travail des développeurs informatiques en leur offrant une architecture "prête à l'emploi" et qui leur permette de ne pas repartir de zéro à chaque nouveau projet.

Les principaux avantages des frameworks sont donc :

- la réutilisation des codes
- la standardisation de la programmation
- la formalisation d'une architecture adaptée aux besoins de chaque entreprise
- La rapidité de l'achèvement d'un projet

À noter aussi que les frameworks sont toujours « enrichis » de l'expérience de tous les développements antérieurs.

ETUDE COMPARATIVE D'AUTRES FRAMEWORKS PHP

Laravel

Créé en 2011, Laravel est devenu l'outil privilégié des développeurs. Grâce à sa grande facilité d'utilisation et à l'élégance de sa syntaxe, mais également à l'écosystème très complet qui l'environne (Homestead, Laracasts, Forge), il permet de travailler efficacement.

Ce framework se distingue par la rapidité de sa courbe d'apprentissage, sa documentation très précise et son moteur de template intégré (Blade). La gestion de la base de données s'effectue par migrations (la structure et l'évolution de la base de données sont décrites par des fichiers).

CodeIgniter

Créé en 2006, puis amélioré en 2014, CodeIgniter a su se démarquer une nouvelle fois grâce aux nombreux atouts de sa version 3. Ce Framework est très léger (2 Mo) et affiche d'excellentes performances.

Ses fonctionnalités et son utilisation sont très faciles à comprendre grâce à sa documentation très complète. Sa flexibilité permet aux utilisateurs de profiter d'une grande liberté dans la structuration du code.

YII

Depuis sa création en 2008, YII a acquis une bonne cote de popularité auprès des développeurs, grâce à une progression fonctionnelle. Par ailleurs, sa rapidité et son extensibilité aident les professionnels à travailler plus vite lorsqu'ils développent des applications PHP.

Les DAO permettent d'interagir avec la base de données. La gestion intégrée de l'authentification, le support natif de jQuery et AJAX, l'outil de génération automatique de code et les nombreux modules proposés figurent parmi les atouts les plus intéressants pour les utilisateurs.

Phalcon

Développé en langage C et distribué sous forme d'une extension, ce framework ne peut pas être utilisé sur un hébergement mutualisé standard. D'après de nombreuses études comparatives, Phalcon affiche une très grande performance et il est, jusqu'à présent, le plus rapide, grâce au moteur de template développé en langage C. Ces performances ne requièrent qu'un faible usage de mémoire. Il est doté d'un ORM intégré (PHQL), d'un système de gestion native des micro-applications, avec les différents éléments d'une framework classique (routeur, architecture MVC, formulaires, cache, etc.).

II. Composer

a) Définition

Composer est un logiciel gestionnaire de dépendances libre écrit en PHP. Il permet à ses utilisateurs de déclarer et d'installer les bibliothèques dont le projet principal a besoin. Le développement a débuté en avril 2011 et a donné lieu à une première version sortie le 1er mars 2012. Développé au début par Nils Adermann et Jordi Boggiano (qui continuent encore aujourd'hui à le maintenir), le projet est maintenant disponible sur la plateforme GitHub. Il est ainsi développé par toute une communauté.

III. Symfony

1) Historique

L'agence web française SensioLabs est à l'origine du framework *Sensio Framework*. À force de toujours recréer les mêmes fonctionnalités de gestion d'utilisateurs, gestion ORM, etc., elle a développé ce framework pour ses propres besoins. Comme ces problématiques étaient souvent les mêmes pour d'autres développeurs, le code a été par la suite partagé avec la communauté des développeurs PHP.

Le projet est alors devenu *Symfony* (conformément à la volonté du créateur de conserver les initiales *S* et *F* de *Sensio Framework*), puis *Symfony2* à partir de la version 24. La version 2 de *Symfony* casse la compatibilité avec la branche 1.x.

Le 5 septembre 2017, Symfony passe la barre du milliard de téléchargements.

2) Avantages

Grâce aux normes et conventions que chaque développeur sur un projet Symfony doit respecter, on obtient une organisation solide des fichiers et du code. Les avantages sont multiples :

- Les développeurs Symfony qui maîtrisent le framework pourront facilement intégrer un projet développé à partir du framework, contrairement à un projet développé en PHP “maison”, où il n'y a pas de normes ni règles imposées. Dans ce dernier cas, la phase d’apprentissage et reprise du code existant peut demander un effort conséquent pour le nouveau développeur intégrant le projet.

- Les fichiers doivent respecter une syntaxe particulière et doivent se trouver au bon endroit dans l’arborescence du projet. Cela garantit une facilité de maintenance sur le long terme, les développeurs savent rapidement dans quel fichier il faut aller pour apporter des modifications.

- L’architecture MVC (Modèle Vue Contrôleur) permet de découper le code représentant la logique métier de l’application et le code de présentation des vues. Ainsi, un intégrateur web voir même un webdesigner n’aura aucun mal à intervenir sur la partie présentation (vues) du projet, sans avoir à intervenir sur des fichiers PHP complexes.

- Favorise la réutilisation de code, la création de tests automatisés (tests unitaires avec phpUnit ou Atoum et tests fonctionnels avec phpUnit ou Behat) et le respect des recommandations PHP-FIG (Des recommandations mondiales pour une

meilleure interopérabilité entre les projets web PHP). Symfony permet donc de produire du code de qualité.

- Symfony intègre des mesures de sécurité préventives pour lutter contre les failles et attaques XSS, CSRF et injection SQL. Contrairement à un développement PHP maison où il faut penser systématiquement à protéger chaque requête, formulaire … Symfony embarque systématiquement ces mécanismes de sécurité, sans avoir à les implémenter à chaque fois.

3) Installation

Pour utiliser Symfony 5 sur Windows il faut au préalable télécharger et installer:

- PHP 7 sur https://windows.php.net/download#php-7.4
- Composer sur https://getcomposer.org/Composer-Setup.exe
- git https://git-scm.com/download/win

Enfin Symfony

- Avec le cmd sur Windows on tape la commande qui permet de creer un projet symfony suivi du nom du répertoire dans lequel ce dernier sera installé
- `composer create-project symfony/skeleton projet_symfony`

a) fichier yml

Les fichiers YML contiennent du code source écrit dans le langage de programmation YAML (YAML Ain't Markup Language) et sont utilisés par les développeurs. Le code contenu dans un fichier YML est lisible par l'homme et généralement utilisé pour sérialiser des données. Un fichier YML permet de lire et d'écrire des données indépendamment du langage de programmation. Ainsi, les fichiers YML peuvent être utilisés avec de nombreux autres fichiers contenant du code source écrit via différents langages tels que C, C#, C++, Java, PHP, etc.

Le composant Symfony Yaml analyse les chaînes YAML pour les convertir en tableaux PHP. Il est également capable de convertir des tableaux PHP en chaînes YAML.

YAML est un excellent format pour vos fichiers de configuration. Les fichiers YAML sont aussi expressifs que les fichiers XML et aussi lisibles que les fichiers INI.

Rapide

L'un des objectifs de Symfony Yaml est de trouver le bon équilibre entre vitesse et fonctionnalités. Il prend en charge uniquement les fonctionnalités nécessaires pour gérer les fichiers de configuration. Les fonctionnalités manquantes notables sont: les directives de document, les messages cités sur plusieurs lignes, les collections de blocs compacts et les fichiers multi-documents.

Analyseur réel

Il arbore un véritable analyseur et est capable d'analyser un grand sous-ensemble de la spécification YAML, pour tous vos besoins de configuration. Cela signifie également que l'analyseur est assez robuste, facile à comprendre et assez simple à étendre.

Effacer les messages d'erreur

Chaque fois que vous rencontrez un problème de syntaxe avec vos fichiers YAML, la bibliothèque génère un message utile avec le nom de fichier et le numéro de ligne où le problème s'est produit. Cela facilite beaucoup le débogage.

Support de vidage

Il est également capable de vider des tableaux PHP vers YAML avec la prise en charge des objets et une configuration de niveau en ligne pour de jolies sorties.

Prise en charge des types ¶

Il prend en charge la plupart des types intégrés YAML comme les dates, les entiers, les nombres octaux, les booléens et bien plus encore…

Prise en charge complète des clés de fusion ¶

Prise en charge complète des références, des alias et de la clé de fusion complète. Ne vous répétez pas en référençant les bits de configuration courants.

Utilisation du composant Symfony YAML ¶

Le composant Symfony Yaml se compose de deux classes principales: l'une analyse les chaînes YAML (`Parser`) et l'autre sauvegarde un tableau PHP dans une chaîne YAML (`Dumper`).

En plus de ces deux classes, la `Yaml`classe agit comme un wrapper fin qui simplifie les utilisations courantes.

Lecture du contenu YAML ¶

La `parse()` méthode analyse une chaîne YAML et la convertit en un tableau PHP:

```php
use Symfony\Component\Yaml\Yaml;

$value = Yaml::parse("foo: bar");
// $value = ['foo' => 'bar']
```

Si une erreur se produit pendant l'analyse, l'analyseur lève une `ParseException` exception indiquant le type d'erreur et la ligne de la chaîne YAML d'origine où l'erreur s'est produite:

```php
use Symfony\Component\Yaml\Exception\ParseException;

try {
    $value = Yaml::parse('...');
} catch (ParseException $exception) {
    printf('Unable to parse the YAML string: %s', $exception->getMessage());
}
```

Lecture de fichiers YAML ¶

La `parseFile()` méthode analyse le contenu YAML du chemin de fichier donné et les convertit en valeur PHP:

Écrire des fichiers YAML ¶

La `dump()` méthode vide n'importe quel tableau PHP dans sa représentation YAML:

```php
use Symfony\Component\Yaml\Yaml;

$array = [
    'foo' => 'bar',
    'bar' => ['foo' => 'bar', 'bar' => 'baz'],
];

$yaml = Yaml::dump($array);

file_put_contents('/path/to/file.yaml', $yaml);
```

Pour l'installer il faut taper cette commande

```
composer require symfony/yaml
```

b) Structure d'un projet Symfony

bin/ : contenant deux executables, la console de Symfony et phpunit

config/ : contenant les fichiers de configuration (routes, ORM...)

public/ : seul dossier accessible de l'extérieur (contenant le contrôleur frontal index.php)

src/ : contenant les fichiers sources de l'application (contrôleurs, entités, formulaires,DAO...)

templates/ : contenant les vues (vue partielle) de l'application

tests/ : contenant les fichiers permettant de tester l'application

translations/ : contenant les fichiers de l'internationalisation

var/ : utilisé par Symfony pendant l'exécution, contenant les données de cache, le log et les sessions

vendor/ : contenant les fichiers nécessaires pour une application Symfony (mentionnés dans composer.json)

Symfony Flex

Flex est un outil Symfony qui permet d'ajouter des nouvelles briques en déployant le moins d'effort possible. En effet, l'ajout d'une brique dans une application nécessite souvent d'ajouter un minimum de configuration, d'enregistrer des nouvelles routes, etc. Nous verrons tout cela en détail pour bien comprendre comment cela fonctionne. Mais sachez que Flex permet d'automatiser pour gagner du temps.

Il est basé sur les recettes Symfony , qui sont un ensemble d'instructions automatisées pour intégrer des packages tiers dans les applications Symfony

Flex fonctionne comme un plugin Composer.

- **Bundle**

Un Bundle est un répertoire dans un projet symfony qui intègre une structure bien définie, ce répertoire permet d'implémenter plusieurs fonctionnalités qui peuvent être utilisées dans d'autre projet symfony. On peut voir un Bundle comme un plug-in dans symfony. Ce framework propose des milliers de Bundles réutilisables quasiment dans tous les projets symfony qu'on peut avoir. il existe même des sites qui proposent de télécharger des Bundles, il y'a par exemple : KnpBundle. Pour créer un bundle on peut procéder de trois manières:

Via Composer en utilisant cette commande

$ composer.phar require nomdubundle/exemple-bundle "version"

La structure de répertoires d'un bundle est destinée à aider à maintenir la cohérence du code entre tous les bundles Symfony. Il suit un ensemble de conventions, mais est flexible pour être ajusté si nécessaire:

- Controller/

Contient les contrôleurs du bundle (par exemple RandomController.php).

- DependencyInjection/

Contient certaines classes d'extension d'injection de dépendance, qui peuvent importer la configuration du service, enregistrer les passes du compilateur ou plus (ce répertoire n'est pas nécessaire).

- Resources/config/

Configuration de maisons, y compris la configuration de routage (par exemple routing.yaml).

- Resources/views/

Contient des modèles organisés par nom de contrôleur (par exemple Random/index.html.twig).

- Resources/public/

Contient des ressources Web (images, feuilles de style, etc.) et est copié ou lié symboliquement dans le public/répertoire du projet via la assets:installcommande console.

- Tests/

Contient tous les tests du bundle.

Un bundle peut être aussi petit ou grand que la fonctionnalité qu'il implémente. Il ne contient que les fichiers dont vous avez besoin et rien d'autre.

c) Commande de génération

- Controller

La commande `generate:controller` génère un nouveau contrôleur comprenant des actions, des tests, des modèles et un routage.

Par défaut, la commande est exécutée en mode interactif et pose des questions pour déterminer le nom, l'emplacement, le format de configuration et la structure par défaut du bundle:

```
$ php bin/console generate:controller
```

--controller

Le nom du contrôleur donné sous forme de notation de raccourci contenant le nom du bundle dans lequel se trouve le contrôleur et le nom du contrôleur (par exemple, AcmeBlogBundle:Postcrée une PostControllerclasse à l'intérieur du AcmeBlogBundlebundle):

$ php bin/console generate:controller --controller=AcmeBlogBundle:Post

- Entity

La `generate:doctrine:entity` commande génère un nouveau stub d'entité Doctrine comprenant la définition de mappage et les propriétés de classe, les getters et les setters.

Par défaut, la commande est exécutée en mode interactif et pose des questions pour déterminer le nom, l'emplacement, le format de configuration et la structure par défaut du bundle:

```
$ php bin/console generate:doctrine:entity
```

- **Relation**

Le nom d'entité donné sous forme de raccourci contenant le nom du bundle dans lequel l'entité est située et le nom de l'entité (par exemple, `AcmeBlogBundle:Post`):

- Maker

  Symfony Maker vous aide à créer des commandes vides, des contrôleurs, des classes de formulaire, des tests et plus encore afin que vous puissiez oublier d'écrire du code passe-partout. Ce bundle est une alternative à SensioGeneratorBundle pour les applications Symfony modernes et nécessite l'utilisation de Symfony 3.4 ou plus récent. Ce bundle suppose que vous utilisez une structure de répertoire Symfony 4 standard, mais de nombreuses commandes peuvent générer du code dans n'importe quelle application.

  ```
  composer require symfony/maker-bundle --dev
  ```

  e) Routing (yml,annotation)

  Les itinéraires peuvent être configurés en YAML, XML, PHP ou en utilisant des annotations. Tous les formats offrent les mêmes fonctionnalités et performances, alors choisissez votre favori.

  ```
  composer require annotations
  ```

  f) ORM: Doctrine

  Symfony fournit tous les outils dont vous avez besoin pour utiliser des bases de données dans vos applications grâce à Doctrine , le meilleur ensemble de bibliothèques PHP pour travailler avec des bases de données. Ces outils prennent en charge les bases de données relationnelles comme MySQL et PostgreSQL ainsi que les bases de données NoSQL comme MongoDB.
  Tout d'abord, installez le support Doctrine via le `orm` pack Symfony , ainsi que le MakerBundle, qui vous aidera à générer du code:

  ```
  composer require symfony/orm-pack
  ```

  ```
  composer require --dev symfony/maker-bundle
  ```

  Les informations de connexion à la base de données sont stockées sous la forme d'une variable d'environnement appelée `DATABASE_URL`. Pour le développement, vous pouvez trouver et personnaliser cela à l'intérieur `.env`:
  Maintenant que vos paramètres de connexion sont configurés, Doctrine peut créer la `db_name` base de données pour vous:

  ```
  php bin/console doctrine:database:create
  ```

Création d'une classe d'entité ¶

Supposons que vous créez une application dans laquelle les produits doivent être affichés. Sans même penser à Doctrine ou aux bases de données, vous savez déjà que vous avez besoin d'un `Product`objet pour représenter ces produits.

Vous pouvez utiliser la `make:entity`commande pour créer cette classe et tous les champs dont vous avez besoin. La commande vous posera quelques questions - répondez-y comme ci-dessous:

```
$ php bin/console make:entity

Class name of the entity to create or update:
> Product

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

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

Field length [255]:
> 255

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

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

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

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

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Woh! Vous avez maintenant un nouveau `src/Entity/Product.php` fichier:

```php
// src/Entity/Product.php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ProductRepository")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

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

    /**
     * @ORM\Column(type="integer")
     */
    private $price;

    public function getId()
    {
        return $this->id;
    }
```

- `Migration`

  La `Product`classe est entièrement configurée et prête à être enregistrée dans une `product`table. Si vous venez de définir cette classe, votre base de données n'a pas encore la `product` table. Pour l'ajouter, vous pouvez exploiter le DoctrineMigrationsBundle , qui est déjà installé:

  ```
  php bin/console make:migration
  ```

  Si vous ouvrez ce fichier, il contient le SQL nécessaire pour mettre à jour votre base de données! Pour exécuter ce SQL, exécutez vos migrations:

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

- `Entity Mananger`

Vous devez activer les extensions pour chaque gestionnaire d'entités pour lequel vous souhaitez activer les extensions. L'ID est l'ID de la connexion DBAL lors de l'utilisation des comportements ORM. Il s'agit de l'ID du gestionnaire de documents lors de l'utilisation de mongoDB.

Ce bundle nécessite un environnement local par défaut utilisé si la traduction n'existe pas dans la langue demandée. Si vous ne le fournissez pas explicitement, il sera par défaut `en`.

```yaml
# app/config/config.yml
# (or config/packages/stof_doctrine_extensions.yaml if you use Flex)
stof_doctrine_extensions:
    default_locale: en_US

    # Only used if you activated the Uploadable extension
    uploadable:
        # Default file path: This is one of the three ways you can configure the path
        default_file_path:       "%kernel.project_dir%/public/uploads"

        # Mime type guesser class: Optional. By default, we provide an adapter for the
        mime_type_guesser_class: Stof\DoctrineExtensionsBundle\Uploadable\MimeTypeGues

        # Default file info class implementing FileInfoInterface: Optional. By default
        default_file_info_class: Stof\DoctrineExtensionsBundle\Uploadable\UploadedFile
    orm:
        default: ~
    mongodb:
        default: ~
```

- persist

  Cette méthode signale à Doctrine que l'objet doit être enregistré. Elle ne doit être utilisée que pour un nouvel objet et non pas pour une mise à jour.

  // Crée l'article et le signale à Doctrine.

  $article1 = **new** Article;

  $article1->setTitre('Mon dernier weekend');

  $entityManager->persist($article);

Doctrine Query Language

DQL signifie Doctrine Query Language et est un dérivé de Object Query Language qui est très similaire au Hibernate Query Language (HQL) ou au Java Persistence Query Language (JPQL).

En substance, DQL offre de puissantes capacités d'interrogation sur votre modèle d'objet. Imaginez tous vos objets qui traînent dans un espace de stockage (comme une base de données d'objets). Lorsque vous écrivez des requêtes DQL, pensez à interroger ce stockage pour choisir un certain sous-ensemble de vos objets.

Types de requêtes DQL

DQL en tant que langage de requête a des constructions SELECT, UPDATE et DELETE qui sont mappées à leurs types d'instructions SQL correspondants. Les instructions INSERT ne sont pas autorisées dans DQL, car les entités et leurs relations doivent être introduites dans le contexte de persistance `EntityManager#persist()` pour garantir la cohérence de votre modèle objet.

Les instructions DQL SELECT sont un moyen très puissant de récupérer des parties de votre modèle de domaine qui ne sont pas accessibles via des associations. De plus, ils vous permettent de récupérer des entités et leurs associations dans une seule instruction SQL select, ce qui peut faire une énorme différence dans les performances par rapport à l'utilisation de plusieurs requêtes.

Les instructions DQL UPDATE et DELETE offrent un moyen d'exécuter des modifications en masse sur les entités de votre modèle de domaine. Cela est souvent nécessaire lorsque vous ne pouvez pas charger toutes les entités affectées d'une mise à jour en bloc dans la mémoire.

SELECT requêtes

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.age > 20');
$users = $query->getResult();
```

- `u` est une soi-disant variable d'identification ou alias qui fait référence à la `MyProject\Model\User` classe. En plaçant cet alias dans la clause SELECT, nous spécifions que nous voulons que toutes les instances de la classe User qui correspondent à cette requête apparaissent dans le résultat de la requête.

- Le mot clé FROM est toujours suivi d'un nom de classe complet qui est à son tour suivi d'une variable d'identification ou d'un alias pour ce nom de classe. Cette classe désigne une racine de notre requête à partir de laquelle nous pouvons naviguer plus loin via des jointures (expliquées plus loin) et des expressions de chemin.
- L'expression `u.age` dans la clause WHERE est une expression de chemin. Les expressions de chemin dans DQL sont facilement identifiables grâce à l'utilisation du '.' opérateur utilisé pour construire des chemins. L'expression de chemin `u.age` fait référence au `age` champ de la classe User.

Joins

Une requête SELECT peut contenir des jointures. Il existe 2 types de jointures: les jointures régulières et les jointures d'extraction.

Jointures régulières : utilisées pour limiter les résultats et / ou calculer les valeurs agrégées.

Récupérer les jointures : En plus des utilisations des jointures régulières: Utilisé pour récupérer les entités liées et les inclure dans le résultat hydraté d'une requête.

Il n'y a pas de mot-clé DQL spécial qui distingue une jointure régulière d'une jointure d'extraction. Une jointure (que ce soit une jointure interne ou externe) devient une jointure d'extraction dès que les champs de l'entité jointe apparaissent dans la partie SELECT de la requête DQL en dehors d'une fonction d'agrégation. Sinon, c'est une jointure régulière.

Exemple:

Adhésion régulière de l'adresse:

```
<?php

$query = $em->createQuery("SELECT u FROM User u JOIN u.address a
WHERE a.city = 'Berlin'");

$users = $query->getResult();
```

Récupérer la jointure de l'adresse:

```
<?php

$query = $em->createQuery("SELECT u, a FROM User u JOIN u.address a
WHERE a.city = 'Berlin'");

$users = $query->getResult();
```

Lorsque Doctrine hydrate une requête avec fetch-join, elle renvoie la classe dans la clause FROM au niveau racine du tableau de résultats. Dans l'exemple précédent, un tableau d'instances User est renvoyé et l'adresse de chaque utilisateur est extraite et hydratée dans la `User#address`variable. Si vous accédez à l'adresse, Doctrine n'a pas besoin de charger paresseusement l'association avec une autre requête.

Paramètres nommés et positionnels

DQL prend en charge les paramètres nommés et positionnels, mais contrairement à de nombreux dialectes SQL, les paramètres positionnels sont spécifiés avec des nombres, par exemple? 1,? 2, etc. Les paramètres nommés sont spécifiés avec: nom1,: nom2 et ainsi de suite.

Lors du référencement, les paramètres des paramètres `Query#setParameter($param, $value)` nommés et positionnels sont utilisés sans leurs préfixes.

Exemples DQL SELECT

Cette section contient un grand ensemble de requêtes DQL et quelques explications sur ce qui se passe. Le résultat réel dépend également du mode d'hydratation.

Hydrate toutes les entités utilisateur:

```
<?php

$query = $em->createQuery('SELECT u FROM MyProject\Model\User u
WHERE u.age > 20');

$users = $Récupérez les ID de tous les utilisateurs Cms:
```

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u');

$ids = $query->getResult(); // array of CmsUser ids
```

Récupérez les ID de tous les utilisateurs qui ont écrit un article:

```
<?php

$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user u');

$ids = $query->getResult(); // array of CmsUser ids
```

Récupérez tous les articles et triez-les par le nom de l'instance d'utilisateurs d'articles:

```
<?php

$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY u.name ASC');

$articles = $query->getResult(); // array of CmsArticle objects
```

Récupérez le nom d'utilisateur et le nom d'un utilisateur Cms:

```
<?php
```

$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');

$users = $query->getResult(); // array of CmsUser username and name values

echo $users[0]['username'];

Récupérez un ForumUser et sa seule entité associée:

<?php

$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');

$users = $query->getResult(); // array of ForumUser objects with the avatar association loaded

echo get_class($users[0]->getAvatar());

Récupérez un CmsUser et récupérez tous les numéros de téléphone qu'il possède:

<?php

$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');

$users = $query->getResult(); // array of CmsUser objects with the phonenumbers association loaded

$phonenumbers = $users[0]->getPhonenumbers();

Hydrate un résultat en Ascendant:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id
ASC');

$users = $query->getResult(); // array of ForumUser objects
```

Ou par ordre décroissant:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id
DESC');

$users = $query->getResult(); // array of ForumUser objects
```

Utilisation des fonctions d'agrégation:

```
<?php

$query = $em->createQuery('SELECT COUNT(u.id) FROM Entities\User u');

$count = $query->getSingleScalarResult();


$query = $em->createQuery('SELECT u, count(g.id) FROM Entities\User u JOIN
u.groups g GROUP BY u.id');

$result = $query->getResult();
```

Avec la clause WHERE et le paramètre positionnel:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');

$query->setParameter(1, 321);

$users = $query->getResult(); // array of ForumUser objects
```

Avec la clause WHERE et le paramètre nommé:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u WHERE
u.username = :name');

$query->setParameter('name', 'Bob');

$users = $query->getResult(); // array of ForumUser objects
```

Avec conditions imbriquées dans la clause WHERE:

```
<?php

$query = $em->createQuery('SELECT u FROM ForumUser u WHERE
(u.username = :name OR u.username = :name2) AND u.id = :id');

$query->setParameters(array(

   'name' => 'Bob',

   'name2' => 'Alice',
```

```
    'id' => 321,
));
$users = $query->getResult(); // array of ForumUser objects
```

Avec COUNT DISTINCT:

```
<?php
$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // array of ForumUser objects
```

Avec l'expression arithmétique dans la clause WHERE:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id + 3) < 10000000');
$users = $query->getResult(); // array of ForumUser objects
```

Récupérez les entités utilisateur avec l'expression arithmétique dans la clause ORDER, à l'aide du HIDDENmot clé:

```
<?php
$query = $em->createQuery('SELECT u, u.posts_count + u.likes_count AS HIDDEN score FROM CmsUser u ORDER BY score');
```

$users = $query->getResult(); // array of User objects

Utilisation d'un LEFT JOIN pour hydrater tous les identifiants utilisateur et les identifiants d'article éventuellement associés:

<?php

$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT JOIN u.articles a');

$results = $query->getResult(); // array of user ids and every article_id for each user

Restreindre une clause JOIN par des conditions supplémentaires spécifiées par WITH:

<?php

$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH a.topic LIKE :foo");

$query->setParameter('foo', '%foo%');

$users = $query->getResult();

Utilisation de plusieurs Fetch JOINs:

<?php

$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a JOIN u.phonenumbers p JOIN a.comments c');

$users = $query->getResult();

ENTRE la clause WHERE:

<?php

$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id BETWEEN ?1 AND ?2');

$query->setParameter(1, 123);

$query->setParameter(2, 321);

$usernames = $query->getResult();

Fonctions DQL dans la clause WHERE:

<?php

$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) = 'someone'");

$usernames = $query->getResult();

IN () Expression:

<?php

$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id IN(46)');

```
$usernames = $query->getResult();


$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id IN (1,
2)');

$users = $query->getResult();


$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id NOT IN
(1)');

$users = $query->getResult();
```

Fonction DQL CONCAT ():

```
<?php

$query = $em->createQuery("SELECT u.id FROM CmsUser u WHERE
CONCAT(u.name, 's') = ?1");

$query->setParameter(1, 'Jess');

$ids = $query->getResult();


$query = $em->createQuery('SELECT CONCAT(u.id, u.name) FROM CmsUser u
WHERE u.id = ?1');

$query->setParameter(1, 321);

$idUsernames = $query->getResult();
```

Clause EXISTS in WHERE avec sous-requête corrélée

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS
(SELECT p.phonenumber FROM CmsPhonenumber p WHERE p.user = u.id)');

$ids = $query->getResult();
```

Obtenez tous les utilisateurs membres de $ group.

```
<?php

$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId
MEMBER OF u.groups');

$query->setParameter('groupId', $group);

$ids = $query->getResult();
```

Obtenez tous les utilisateurs qui ont plus d'un numéro de téléphone

```
<?php

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE
SIZE(u.phonenumbers) > 1');

$users = $query->getResult();
```

Obtenez tous les utilisateurs qui n'ont pas de numéro de téléphone

<?php

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS EMPTY');

$users = $query->getResult();

Obtenez toutes les instances d'un type spécifique, à utiliser avec les hiérarchies d'héritage:query->getResult();

FORMS

- **Contraintes**

  La création et le traitement de formulaires HTML sont difficiles et répétitifs. Vous devez gérer le rendu des champs de formulaire HTML, la validation des données soumises, le mappage des données de formulaire en objets et bien plus encore. Symfony comprend une puissante fonctionnalité de formulaire qui fournit toutes ces fonctionnalités et bien d'autres pour des scénarios vraiment complexes.

Installation

Dans les applications utilisant Symfony Flex , exécutez cette commande pour installer la fonctionnalité de formulaire avant de l'utiliser:

```
composer require symfony/form
```

Utilisation

Le flux de travail recommandé lors de l'utilisation de formulaires Symfony est le suivant:

1. **Créez le formulaire** dans un contrôleur Symfony ou en utilisant une classe de formulaire dédiée;
2. **Rendez le formulaire** dans un modèle afin que l'utilisateur puisse le modifier et le soumettre;
3. **Traitez le formulaire** pour valider les données soumises, transformez-les en données PHP et faites-en quelque chose (par exemple, conservez-les dans une base de données).

Chacune de ces étapes est expliquée en détail dans les sections suivantes. Pour rendre les exemples plus faciles à suivre, ils supposent tous que vous créez une application de liste Todo simple qui affiche des «tâches».

Les utilisateurs créent et modifient des tâches à l'aide de formulaires Symfony. Chaque tâche est une instance de la `Task`classe suivante :

```php
// src/Entity/Task.php
namespace App\Entity;

class Task
{
    protected $task;
    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }

    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }

    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Cette classe est un «objet PHP simple et ancien» car, jusqu'à présent, elle n'a rien à voir avec Symfony ou toute autre bibliothèque. C'est un objet PHP normal qui résout directement un problème à l'intérieur de *votre* application (c'est-à-dire la nécessité de représenter une tâche dans votre application). Mais vous pouvez également modifier les entités Doctrine de la même manière.

Types de formulaires ¶

Avant de créer votre premier formulaire Symfony, il est important de comprendre le concept de «type de formulaire». Dans d'autres projets, il est courant de faire la différence entre «formulaires» et «champs de formulaire». Dans Symfony, tous sont des «types de formulaires»:

- un seul champ de formulaire est un «type de formulaire» (par exemple);`<input type="text">TextType`
- un groupe de plusieurs champs HTML utilisés pour saisir une adresse postale est un «type de formulaire» (par exemple `PostalAddressType`);
- un ensemble `<form>`avec plusieurs champs pour modifier un profil utilisateur est un «type de formulaire» (par exemple `UserProfileType`).

Cela peut être déroutant au début, mais cela vous semblera naturel assez tôt. En outre, il simplifie le code et facilite la mise en œuvre des champs de formulaire de «composition» et «d'intégration».

Symfony propose des dizaines de types de formulaires et vous pouvez également créer vos propres types de formulaires .

Création de formulaires

Symfony fournit un objet «form builder» qui vous permet de décrire les champs du formulaire à l'aide d'une interface fluide. Plus tard, ce générateur crée l'objet de formulaire réel utilisé pour rendre et traiter le contenu.

Création de formulaires dans les contrôleurs

Si votre contrôleur s’étend du AbstractController , utilisez l’ `createFormBuilder()`assistant:

```php
// src/Controller/TaskController.php
namespace App\Controller;

use App\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\HttpFoundation\Request;

class TaskController extends AbstractController
{
    public function new(Request $request)
    {
        // creates a task object and initializes some data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save', SubmitType::class, ['label' => 'Create Task'])
            ->getForm();

        // ...
    }
}
```

Si votre contrôleur ne s'étend pas depuis `AbstractController`, vous devrez récupérer les services dans votre contrôleur et utiliser la `createBuilder()` méthode du `form.factory` service.

Dans cet exemple, vous avez ajouté deux champs à votre formulaire - `task` et `dueDate` - correspondant aux propriétés `task` et `dueDate` de la `Task` classe. Vous avez également attribué à chacun un type de formulaire (par exemple, `TextType` et `DateType`), représenté par son nom de classe complet. Enfin, vous avez ajouté un bouton d'envoi avec une étiquette personnalisée pour soumettre le formulaire au serveur.

Création de classes de formulaires ¶

Symfony recommande de mettre le moins de logique possible dans les contrôleurs. C'est pourquoi il est préférable de déplacer des formulaires complexes vers des classes dédiées plutôt que de les définir dans des actions de contrôleur. De plus, les formulaires définis dans les classes peuvent être réutilisés dans plusieurs actions et services.

Les classes de formulaire sont des types de formulaire qui implémentent `FormTypeInterface`. Cependant, il est préférable d'étendre à partir de `AbstractType`, qui implémente déjà l'interface et fournit certains utilitaires:

```php
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('task', TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save', SubmitType::class)
        ;
    }
}
```

La classe de formulaire contient toutes les instructions nécessaires pour créer le formulaire de tâche. Dans les contrôleurs s'étendant de AbstractController , utilisez l' `createForm()` assistant (sinon, utilisez la `create()` méthode du `form.factory` service):

```php
// src/Controller/TaskController.php
namespace App\Controller;

use App\Form\Type\TaskType;
// ...

class TaskController extends AbstractController
{
    public function new()
    {
        // creates a task object and initializes some data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createForm(TaskType::class, $task);

        // ...
    }
}
```

Chaque formulaire doit connaître le nom de la classe qui contient les données sous-jacentes (par exemple `App\Entity\Task`). Habituellement, cela est juste supposé en fonction de l'objet passé au deuxième argument `createForm()`(c'est- à -dire `$task`). Plus tard, lorsque vous commencerez à incorporer des formulaires , cela ne sera plus suffisant.

Ainsi, bien que cela ne soit pas toujours nécessaire, il est généralement préférable de spécifier explicitement l' `data_class`option en ajoutant ce qui suit à votre classe de type de formulaire:

```php
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use App\Entity\Task;
use Symfony\Component\OptionsResolver\OptionsResolver;
// ...

class TaskType extends AbstractType
{
    // ...

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

Validation des formulaires

Dans la section précédente, vous avez appris comment un formulaire peut être soumis avec des données valides ou non valides. Dans Symfony, la question n'est pas de savoir si le «formulaire» est valide, mais si l'objet sous-jacent (`$task`dans cet exemple) est valide après que le formulaire lui a appliqué les données soumises. L'appel `$form->isValid()`est un raccourci qui demande à l' `$task`objet s'il possède ou non des données valides.

Avant d'utiliser la validation, ajoutez-en la prise en charge dans votre application:

```
composer require symfony/validator
```

La validation se fait en ajoutant un ensemble de règles (appelées contraintes) à une classe. Pour voir cela en action, ajoutez des contraintes de validation afin que le `task`champ ne puisse pas être vide et le `dueDate`champ ne puisse pas être vide et doit être un objet DateTime valide.

```php
// src/Entity/Task.php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank
     */
    public $task;

    /**
     * @Assert\NotBlank
     * @Assert\Type("\DateTime")
     */
    protected $dueDate;
}
```

Sécurité

Dans les applications utilisant Symfony Flex , exécutez cette commande pour installer la fonction de sécurité avant de l'utiliser:

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

```yaml
# config/packages/security.yaml
security:
    enable_authenticator_manager: true
    # ...
```

Créez votre classe d'utilisateurs

Peu importe *comment* vous vous authentifierez (par exemple, formulaire de connexion ou jetons d'API) ou *où* vos données d'utilisateur seront stockées (base de données, authentification unique), l'étape suivante est toujours la même: créez une classe «Utilisateur». La façon la plus simple est d'utiliser le MakerBundle .

Supposons que vous souhaitiez stocker vos données utilisateur dans la base de données avec Doctrine:

```
> php bin/console make:user

The name of the security user class (e.g. User) [User]:
> User

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g.
email, username, uuid [email]
> email

Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

C'est tout! La commande pose plusieurs questions afin de générer exactement ce dont vous avez besoin. Le plus important est le `User.php`fichier lui-même. La *seule* règle concernant votre `User`classe est qu'elle *doit être* implémentée `UserInterface`. N'hésitez pas à ajouter *tout* autre champ ou logique dont vous avez besoin. Si votre `User`classe est une entité (comme dans cet exemple), vous pouvez utiliser la commande make: entity pour ajouter plus de champs. Assurez-vous également d'effectuer et d'exécuter une migration pour la nouvelle entité:

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

Le «fournisseur d'utilisateur»

En plus de votre `User`classe, vous avez également besoin d'un «fournisseur d'utilisateur»: une classe qui aide avec quelques choses, comme le rechargement des données utilisateur de la session et certaines fonctionnalités optionnelles, comme se souvenir de moi et l' emprunt d'identité .

Heureusement, la `make:user`commande en a déjà configuré un dans votre `security.yaml`fichier sous la `providers`clé.

Si votre `User`classe est une entité, vous n'avez rien d'autre à faire. Mais si votre classe n'est *pas* une entité, elle `make:user`aura également généré une `UserProvider`classe que vous devez terminer. En savoir plus sur les fournisseurs d'utilisateurs ici: Fournisseurs d'utilisateurs .

Encodage des mots de passe

Toutes les applications n'ont pas «d'utilisateurs» qui ont besoin de mots de passe. *Si* vos utilisateurs ont des mots de passe, vous pouvez contrôler la façon dont ces mots de passe sont encodés `security.yaml`. La `make:user` commande pré-configurera ceci pour vous:

```yaml
# config/packages/security.yaml
security:
    # ...

    encoders:
        # use your user class name here
        App\Entity\User:
            # Use native password encoder
            # This value auto-selects the best possible hashing algorithm
            # (i.e. Sodium when available).
            algorithm: auto
```

Maintenant que Symfony sait *comment* vous voulez encoder les mots de passe, vous pouvez utiliser le `UserPasswordEncoderInterface`service pour ce faire avant d'enregistrer vos utilisateurs dans la base de données.

Par exemple, en utilisant DoctrineFixturesBundle , vous pouvez créer des utilisateurs de base de données factices:

php bin/console make:fixtures

The class name of the fixtures to create (e.g. AppFixtures):

> UserFixtures

Utilisez ce service pour coder les mots de passe:

```
// src/DataFixtures/UserFixtures.php


+ use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

// ...


class UserFixtures extends Fixture

{

+     private $passwordEncoder;


+     public function __construct(UserPasswordEncoderInterface $passwordEncoder)

+     {

+         $this->passwordEncoder = $passwordEncoder;

+     }


    public function load(ObjectManager $manager)

    {

        $user = new User();

        // ...
```

```
+        $user->setPassword($this->passwordEncoder->encodePassword(
+            $user,
+            'the_new_password'
+        ));


        // ...
    }
}
```

Vous pouvez encoder manuellement un mot de passe en exécutant:

```
 php bin/console security:encode-password
```

3a) Authentification et pare-feu ¶

Nouveau dans la version 5.1:L' option a été introduite dans Symfony 5.1. Avant la version 5.1, il était activé à l'aide delazy: trueanonymous: lazy

Le système de sécurité est configuré dans config/packages/security.yaml. La section la plus importante est firewalls:

YAML

```
# config/packages/security.yaml
```

```
security:
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            lazy: true
```

XML

PHP

Un «pare-feu» est votre système d'authentification: la configuration ci-dessous définit comment vos utilisateurs pourront s'authentifier (par exemple, formulaire de connexion, jeton d'API, etc.).

Un seul pare-feu est actif sur chaque demande: Symfony utilise la patternclé pour trouver la première correspondance (vous pouvez également faire une correspondance par hôte ou autre). Le devpare-feu est vraiment un faux pare-feu: il s'assure simplement que vous ne bloquez pas accidentellement les outils de développement de Symfony - qui vivent sous des URL comme /_profiler et /_wdt.

Toutes les URL réelles sont gérées par le mainpare - feu (aucune patternclé ne signifie qu'elle correspond à toutes les URL). Un pare-feu peut avoir de nombreux modes d'authentification, c'est-à-dire de nombreuses façons de poser la question

«Qui êtes-vous?». Souvent, l'utilisateur est inconnu (c'est-à-dire non connecté) lors de sa première visite sur votre site Web. Le anonymousmode, s'il est activé, est utilisé pour ces demandes.

En fait, si vous allez sur la page d'accueil en ce moment, vous y aurez accès et vous verrez que vous êtes "authentifié" en tant que anon.. Le pare-feu a vérifié qu'il ne connaissait pas votre identité et vous êtes donc anonyme:

_images / anonymous_wdt.png

Cela signifie que toute demande peut avoir un jeton anonyme pour accéder à certaines ressources, tandis que certaines actions (c'est-à-dire certaines pages ou boutons) peuvent toujours nécessiter des privilèges spécifiques. Un utilisateur peut alors accéder à une connexion de formulaire sans être authentifié en tant qu'utilisateur unique (sinon une boucle de redirection infinie se produirait demandant à l'utilisateur de s'authentifier tout en essayant de le faire).

Vous apprendrez plus tard comment refuser l'accès à certaines URL, contrôleurs ou parties de modèles.

POINTE

Le lazymode anonyme empêche le démarrage de la session s'il n'y a pas besoin d'autorisation (c.-à-d. Vérification explicite d'un privilège d'utilisateur). Ceci est important pour garder les requêtes en cache (voir Cache HTTP).

REMARQUE

Si vous ne voyez pas la barre d'outils, installez le profileur avec:

composer require --dev symfony/profiler-pack

Maintenant que nous comprenons notre pare-feu, l'étape suivante consiste à créer un moyen pour vos utilisateurs de s'authentifier!

3b) Authentification de vos utilisateurs ¶

L'authentification dans Symfony peut sembler un peu «magique» au début. En effet, au lieu de créer une route et un contrôleur pour gérer la connexion, vous activerez un fournisseur d'authentification : du code qui s'exécute automatiquement avant l' appel de votre contrôleur.

Symfony a plusieurs fournisseurs d'authentification intégrés . Si votre cas d'utilisation correspond exactement à l' un d'eux , tant mieux ! Mais, dans la plupart des cas - y compris un formulaire de connexion - nous vous recommandons de créer un Guard Authenticator : une classe qui vous permet de contrôler chaque partie du processus d'authentification (voir la section suivante).

POINTE

Si votre application connecte les utilisateurs via un service tiers tel que Google, Facebook ou Twitter (connexion sociale), consultez le bundle de la communauté HWIOAuthBundle .

Authentificateurs de garde ¶

Un authentificateur Guard est une classe qui vous donne un contrôle complet sur votre processus d'authentification. Il existe de nombreuses façons différentes de créer un authentificateur; voici quelques cas d'utilisation courants:

Comment créer un formulaire de connexion

Système d'authentification personnalisé avec garde (exemple de jeton d'API) - voir ceci pour la description la plus détaillée des authentificateurs et comment ils fonctionnent

4) Refuser l'accès, les rôles et autres autorisations ¶

Les utilisateurs peuvent désormais se connecter à votre application en utilisant votre formulaire de connexion. Génial! Maintenant, vous devez apprendre à refuser l'accès et à travailler avec l'objet User. C'est ce qu'on appelle l' autorisation , et son travail consiste à décider si un utilisateur peut accéder à une ressource (une URL, un objet modèle, un appel de méthode,…).

Le processus d'autorisation a deux aspects différents:

L'utilisateur reçoit un ensemble spécifique de rôles lors de la connexion (par exemple ROLE_ADMIN).

Vous ajoutez du code pour qu'une ressource (par exemple URL, contrôleur) nécessite un «attribut» spécifique (le plus souvent un rôle ROLE_ADMIN) pour être accessible.

Rôles ¶

Lorsqu'un utilisateur se connecte, Symfony appelle la getRoles()méthode sur votre User objet pour déterminer les rôles de cet utilisateur. Dans la Userclasse que nous avons générée précédemment, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit toujours attribuer au moins un rôle ROLE_USER::

```
// src/Entity/User.php


// ...
class User
{
    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];


    // ...
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';


        return array_unique($roles);
    }
}
```

C'est une belle valeur par défaut, mais vous pouvez faire ce que vous voulez pour déterminer les rôles qu'un utilisateur doit avoir. Voici quelques directives:

Chaque rôle doit commencer par ROLE_ (sinon, les choses ne fonctionneront pas comme prévu)

Autre que la règle ci-dessus, un rôle n'est qu'une chaîne et vous pouvez inventer ce dont vous avez besoin (par exemple ROLE_PRODUCT_ADMIN).

Vous utiliserez ces rôles à côté pour accorder l'accès à des sections spécifiques de votre site. Vous pouvez également utiliser une hiérarchie de rôles où certains rôles vous donnent automatiquement d'autres rôles.

Ajouter du code pour refuser l'accès ¶

Il existe deux façons de refuser l'accès à quelque chose:

access_control dans security.yaml vous permet de protéger les modèles d'URL (par exemple/admin/*). Plus simple, mais moins flexible;

dans votre contrôleur (ou autre code) .

Sécurisation des modèles d'URL (access_control) ¶

La façon la plus simple de sécuriser une partie de votre application est de sécuriser un modèle d'URL complet dans security.yaml. Par exemple, pour exiger ROLE_ADMINpour toutes les URL commençant par /admin, vous pouvez:

YAML

```yaml
# config/packages/security.yaml
security:
    # ...

    firewalls:
        # ...
        main:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: '^/admin', roles: ROLE_ADMIN }

        # or require ROLE_ADMIN or IS_AUTHENTICATED_FULLY for /admin*
        - { path: '^/admin', roles: [IS_AUTHENTICATED_FULLY, ROLE_ADMIN] }

        # the 'path' value can be any valid regular expression
        # (this one will match URLs like /api/post/7298 and /api/comment/528491)
        - { path: ^/api/(post|comment)/\d+$, roles: ROLE_USER }
```

XML

PHP

Vous pouvez définir autant de modèles d'URL que vous le souhaitez - chacun est une expression régulière. MAIS , une seule sera mise en correspondance par demande: Symfony démarre en haut de la liste et s'arrête lorsqu'il trouve la première correspondance:

YAML

```
# config/packages/security.yaml
security:
    # ...

    access_control:
        # matches /admin/users/*
        - { path: '^/admin/users', roles: ROLE_SUPER_ADMIN }

        # matches /admin/* except for anything matching the above rule
        - { path: '^/admin', roles: ROLE_ADMIN }
```

XML

PHP

Le fait de faire précéder le chemin d'accès ^signifie que seules les URL commençant par le modèle sont mises en correspondance. Par exemple, un chemin d'accès /admin(sans le ^) correspondrait /admin/foomais correspondrait également à des URL comme /foo/admin.

Chacun access_controlpeut également correspondre à l'adresse IP, au nom d'hôte et aux méthodes HTTP. Il peut également être utilisé pour rediriger un utilisateur vers la httpsversion d'un modèle d'URL. Voir Comment fonctionne la sécurité access_control? .

Sécurisation des contrôleurs et autre code ¶

Vous pouvez refuser l'accès depuis l'intérieur d'un contrôleur:

```
// src/Controller/AdminController.php
// ...

public function adminDashboard()
{
    $this->denyAccessUnlessGranted('ROLE_ADMIN');

    // or add an optional message - seen by developers
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without having ROLE_ADMIN');
```

}

C'est tout! Si l'accès n'est pas accordé, un spécial AccessDeniedException est lancé et plus aucun code dans votre contrôleur n'est exécuté. Ensuite, l'une des deux choses se produira:

Si l'utilisateur n'est pas encore connecté, il lui sera demandé de se connecter (par exemple redirigé vers la page de connexion).

Si l'utilisateur est connecté, mais n'a pas le ROLE_ADMINrôle, la page 403 d'accès refusé s'affiche (que vous pouvez personnaliser).

Grâce au SensioFrameworkExtraBundle, vous pouvez également sécuriser votre contrôleur à l'aide d'annotations:

```
// src/Controller/AdminController.php

// ...


+ use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;


+ /**

+  * Require ROLE_ADMIN for *every* controller method in this class.

+  *

+  * @IsGranted("ROLE_ADMIN")

+  */
```

```
class AdminController extends AbstractController
{
+    /**
+     * Require ROLE_ADMIN for only this controller method.
+     *
+     * @IsGranted("ROLE_ADMIN")
+     */
    public function adminDashboard()
    {
        // ...
    }
}
```

Pour plus d'informations, consultez la documentation FrameworkExtraBundle .

Contrôle d'accès dans les modèles ¶

Si vous souhaitez vérifier si l'utilisateur actuel a un certain rôle, vous pouvez utiliser la is_granted()fonction d'assistance intégrée dans n'importe quel modèle Twig:

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

Sécuriser d'autres services ¶

Voir Comment sécuriser un service ou une méthode dans votre application .

Définition des autorisations utilisateur individuelles ¶

La plupart des applications nécessitent des règles d'accès plus spécifiques. Par exemple, un utilisateur ne devrait pouvoir modifier que ses propres commentaires sur un blog. Les électeurs vous permettent d'écrire tout logique métier dont vous avez besoin pour déterminer l' accès. L'utilisation de ces votants est similaire aux contrôles d'accès basés sur les rôles mis en œuvre dans les chapitres précédents. Lisez Comment utiliser les électeurs pour vérifier les autorisations des utilisateurs pour savoir comment implémenter votre propre électeur.

Vérification pour voir si un utilisateur est connecté (IS_AUTHENTICATED_FULLY) ¶

Si vous souhaitez uniquement vérifier si un utilisateur est connecté (vous ne vous souciez pas des rôles), vous avez deux options. Tout d'abord, si vous avez donné à chaque utilisateur ROLE_USER, vous pouvez simplement vérifier ce rôle. Sinon, vous pouvez utiliser un «attribut» spécial à la place d'un rôle:

```
// ...

public function adminDashboard()
{
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // ...
```

}

Vous pouvez utiliser IS_AUTHENTICATED_FULLYn'importe où les rôles sont utilisés: comme access_controlou dans Twig.

IS_AUTHENTICATED_FULLYn'est pas un rôle, mais il agit en quelque sorte comme un rôle, et chaque utilisateur qui s'est connecté l'aura. En fait, il existe des attributs spéciaux comme celui-ci:

IS_AUTHENTICATED_REMEMBERED: Tous les utilisateurs connectés l'ont, même s'ils sont connectés à cause d'un cookie "se souvenir de moi". Même si vous n'utilisez pas la fonctionnalité Se souvenir de moi , vous pouvez l'utiliser pour vérifier si l'utilisateur est connecté.

IS_AUTHENTICATED_FULLY: Ceci est similaire à IS_AUTHENTICATED_REMEMBERED, mais plus fort. Les utilisateurs qui se sont connectés uniquement à cause d'un cookie "se souvenir de moi" auront IS_AUTHENTICATED_REMEMBEREDmais ne l'auront pas IS_AUTHENTICATED_FULLY.

IS_AUTHENTICATED_ANONYMOUSLY: Tous les utilisateurs (même anonymes) l'ont - ceci est utile lors de la mise en liste blanche des URL pour garantir l'accès - certains détails sont dans Comment fonctionne la sécurité access_control? .

IS_ANONYMOUS: Seuls les utilisateurs anonymes sont mis en correspondance par cet attribut.

IS_REMEMBERED: Seuls les utilisateurs authentifiés à l'aide de la fonctionnalité se souvenir de moi , (c'est-à-dire un cookie souvenir de moi).

IS_IMPERSONATOR: Lorsque l'utilisateur actuel emprunte l'identité d' un autre utilisateur dans cette session, cet attribut correspondra.

Nouveau dans la version 5.1:Les IS_ANONYMOUS, IS_REMEMBEREDet les IS_IMPERSONATOR attributs ont été introduits dans Symfony 5.1.

5a) Récupération de l'objet utilisateur ¶

Après l'authentification, l' Userobjet de l'utilisateur actuel est accessible via le getUser()raccourci:

```
public function index()
{
    // usually you'll want to make sure the user is authenticated first
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // returns your User object, or null if the user is not authenticated
    // use inline documentation to tell your editor your exact User class
    /** @var \App\Entity\User $user */
    $user = $this->getUser();

    // Call whatever methods you've added to your User class
    // For example, if you added a getFirstName() method, you can use that.
    return new Response('Well hi there '.$user->getFirstName());
}
```

5b) Récupérer l'utilisateur d'un service ¶

Si vous devez obtenir l'utilisateur connecté à partir d'un service, utilisez le Securityservice:

```
// src/Service/ExampleService.php
// ...

use Symfony\Component\Security\Core\Security;

class ExampleService
{
    private $security;

    public function __construct(Security $security)
    {
        // Avoid calling getUser() in the constructor: auth may not
        // be complete yet. Instead, store the entire Security object.
        $this->security = $security;
    }

    public function someMethod()
```

```
    {
        // returns User object or null if not authenticated
        $user = $this->security->getUser();
    }
}
```

Récupérer l'utilisateur dans un modèle ¶

Dans un modèle Twig, l'objet utilisateur est disponible via la app.uservariable grâce à la variable d'application globale Twig :

```
1
2
3
```

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Email: {{ app.user.email }}</p>
{% endif %}
```

Déconnexion ¶

Pour activer la déconnexion, activez le logoutparamètre config sous votre pare-feu:

YAML

```
# config/packages/security.yaml
security:
    # ...

    firewalls:
        main:
            # ...
            logout:
                path:   app_logout

                # where to redirect after logout
                # target: app_any_route
```

XML

PHP

Ensuite, vous devrez créer un itinéraire pour cette URL (mais pas un contrôleur):

Annotations

```
// src/Controller/SecurityController.php
namespace App\Controller;
```

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

class SecurityController extends AbstractController
{
    /**
     * @Route("/logout", name="app_logout", methods={"GET"})
     */
    public function logout()
    {
        // controller can be blank: it will never be executed!
        throw new \Exception('Don\'t forget to activate logout in security.yaml');
    }
}
```

YAML

XML

PHP

Et c'est tout! En envoyant un utilisateur sur la app_logoutroute (c'est-à-dire vers /logout) Symfony désidentifiera l'utilisateur actuel et le redirigera.

Personnalisation de la déconnexion ¶

Nouveau dans la version 5.1:Le a LogoutEventété introduit dans Symfony 5.1. Avant cette version, vous deviez utiliser un gestionnaire de réussite de déconnexion pour personnaliser la déconnexion.

Dans certains cas, vous devez exécuter une logique supplémentaire lors de la déconnexion (par exemple, invalider certains jetons) ou vouloir personnaliser ce qui se passe après une déconnexion. Lors de la déconnexion, un LogoutEvent est envoyé. Enregistrez un écouteur d'événements ou un abonné pour exécuter une logique personnalisée. Les informations suivantes sont disponibles dans la classe d'événements:

getToken()

Renvoie le jeton de sécurité de la session sur le point de se déconnecter.

getRequest()

Renvoie la demande en cours.

getResponse()

Renvoie une réponse, si elle est déjà définie par un écouteur personnalisé. Utilisez setResponse()pour configurer une réponse de déconnexion personnalisée.

POINTE

Chaque pare-feu de sécurité possède son propre répartiteur d'événements (security.event_dispatcher.FIREWALLNAME). L'événement de déconnexion est distribué à la fois sur le répartiteur global et sur le pare-feu. Vous pouvez vous

inscrire sur le répartiteur de pare-feu si vous souhaitez que votre écouteur soit exécuté uniquement pour un pare-feu spécifique. Par exemple, si vous avez un pare api - mainfeu et , utilisez cette configuration pour vous inscrire uniquement sur l'événement de déconnexion dans le mainpare - feu:

YAML

```
# config/services.yaml
services:
    # ...

    App\EventListener\CustomLogoutSubscriber:
        tags:
            - name: kernel.event_subscriber
              dispatcher: security.event_dispatcher.main
```

XML

PHP

Rôles hiérarchiques ¶

Au lieu de donner plusieurs rôles à chaque utilisateur, vous pouvez définir des règles d'héritage de rôle en créant une hiérarchie de rôles:

YAML

```
# config/packages/security.yaml
security:
    # ...

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

XML

PHP

Les utilisateurs avec le ROLE_ADMINrôle auront également le ROLE_USERrôle. Et les utilisateurs avec ROLE_SUPER_ADMIN, auront automatiquement ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCHet ROLE_USER(héritée de ROLE_ADMIN).

Pour que la hiérarchie des rôles fonctionne, n'essayez pas d'appeler $user->getRoles()manuellement. Par exemple, dans un contrôleur s'étendant du contrôleur de base :

```
// BAD - $user->getRoles() will not know about the role hierarchy
```

```
$hasAccess = in_array('ROLE_ADMIN', $user->getRoles());


// GOOD - use of the normal security methods

$hasAccess = $this->isGranted('ROLE_ADMIN');

$this->denyAccessUnlessGranted('ROLE_ADMIN');
```

Templating

Le lecteur astucieux a remarqué que notre framework codait en dur la façon dont le «code» spécifique (les modèles) est exécuté. Pour les pages simples comme celles que nous avons créées jusqu'à présent, ce n'est pas un problème, mais si vous voulez ajouter plus de logique, vous seriez obligé de mettre la logique dans le modèle lui-même, ce qui n'est probablement pas une bonne idée, surtout si vous ont toujours à l'esprit le principe de la séparation des préoccupations.

Séparons le code du modèle de la logique en ajoutant une nouvelle couche: le contrôleur: la mission du contrôleur est de générer une réponse basée sur les informations transmises par la demande du client.

```
Mod// example.com/web/front.php


// ...

try {
```

```
    $request->attributes->add($matcher->match($request->getPathInfo()));
    $response = call_user_func('render_template', $request);
} catch (Routing\Exception\ResourceNotFoundException $exception) {
    $response = new Response('Not Found', 404);
} catch (Exception $exception) {
    $response = new Response('An error occurred', 500);
}
```

Le rendu étant désormais effectué par une fonction externe (render_template() ici), nous devons lui passer les attributs extraits de l'URL. Nous aurions pu les passer comme argument supplémentaire à render_template(), mais à la place, utilisons une autre fonctionnalité de la Requestclasse appelée attributs : Les attributs de demande sont un moyen de joindre des informations supplémentaires sur la demande qui ne sont pas directement liées aux données de la demande HTTP.

Vous pouvez maintenant créer la render_template()fonction, un contrôleur générique qui rend un modèle en l'absence de logique spécifique. Pour conserver le même modèle que précédemment, les attributs de requête sont extraits avant le rendu du modèle:

```
function render_template($request)
{
```

```
    extract($request->attributes->all(), EXTR_SKIP);
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

    return new Response(ob_get_clean());
}
```

Comme il render_templateest utilisé comme argument de la call_user_func() fonction PHP , nous pouvons la remplacer par n'importe quel rappel PHP valide . Cela nous permet d'utiliser une fonction, une fonction anonyme ou une méthode d'une classe comme contrôleur… votre choix.

Par convention, pour chaque route, le contrôleur associé est configuré via l' _controllerattribut route:

```
$routes->add('hello', new Routing\Route('/hello/{name}', [
    'name' => 'World',
    '_controller' => 'render_template',
]));

try {
```

```
    $request->attributes->add($matcher->match($request->getPathInfo()));

    $response = call_user_func($request->attributes->get('_controller'),
$request);

} catch (Routing\Exception\ResourceNotFoundException $exception) {

    $response = new Response('Not Found', 404);

} catch (Exception $exception) {

    $response = new Response('An error occurred', 500);

}
```

Un itinéraire peut désormais être associé à n'importe quel contrôleur et au sein d'un contrôleur, vous pouvez toujours utiliser le render_template()pour rendre un modèle:

```
$routes->add('hello', new Routing\Route('/hello/{name}', [

    'name' => 'World',

    '_controller' => function ($request) {

        return render_template($request);

    }

]));
```

C'est plutôt flexible car vous pouvez modifier l'objet Response par la suite et vous pouvez même passer des arguments supplémentaires au modèle:

```
$routes->add('hello', new Routing\Route('/hello/{name}', [
    'name' => 'World',
    '_controller' => function ($request) {
        // $foo will be available in the template
        $request->attributes->set('foo', 'bar');

        $response = render_template($request);

        // change some header
        $response->headers->set('Content-Type', 'text/plain');

        return $response;
    }
]));
```

Voici la version mise à jour et améliorée de notre framework:

```
// example.com/web/front.php
```

```
require_once __DIR__.'/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing;

function render_template($request)
{
    extract($request->attributes->all(), EXTR_SKIP);
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

    return new Response(ob_get_clean());
}

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$context->fromRequest($request);
```

```
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);


try {
    $request->attributes->add($matcher->match($request->getPathInfo()));
    $response = call_user_func($request->attributes->get('_controller'),
$request);
} catch (Routing\Exception\ResourceNotFoundException $exception) {
    $response = new Response('Not Found', 404);
} catch (Exception $exception) {
    $response = new Response('An error occurred', 500);
}


$response->send();
```

Pour célébrer la naissance de notre nouveau framework, créons une toute nouvelle application qui nécessite une logique simple. Notre application a une page qui indique si une année donnée est une année bissextile ou non. Lorsque vous appelez /is_leap_year, vous obtenez la réponse pour l'année en cours, mais vous pouvez également spécifier une année comme dans /is_leap_year/2009. Étant générique, le framework n'a pas besoin d'être modifié en aucune façon, créez un nouveau app.phpfichier:

```
// example.com/src/app.php

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\Routing;


function is_leap_year($year = null) {

    if (null === $year) {

        $year = date('Y');

    }


    return 0 === $year % 400 || (0 === $year % 4 && 0 !== $year % 100);

}


$routes = new Routing\RouteCollection();

$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', [

    'year' => null,

    '_controller' => function ($request) {

        if (is_leap_year($request->attributes->get('year'))) {

            return new Response('Yep, this is a leap year!');

        }
```

```
        return new Response('Nope, this is not a leap year.');
    }
]));

return $routes;
```

La is_leap_year()fonction retourne truelorsque l'année est une année bissextile, falseautrement. Si l'année est null, l'année en cours est testée. Le contrôleur fait peu: il obtient l'année des attributs de la requête, la transmet à la is_leap_year()fonction et en fonction de la valeur de retour, il crée un nouvel objet Response.

Comme toujours, vous pouvez décider de vous arrêter ici et d'utiliser le framework tel quel; il est probablement tout ce que vous devez créer des sites Web simples comme ceux de fantaisie d' une page des sites Web et nous espérons que quelques autres.ifiez la partie de rendu de modèle du cadre pour lire comme suit: