Twig

Voir : https://twig.symfony.com/doc/2.x/ and https://symfony.com/doc/current/templating.html

Les contrôleurs sont chargés de gérer chaque requête entrant dans une application Symfony. Habituellement, ils génèrent un template qui retournera les contenus de la réponse.

Un template est simplement un fichier texte qui peut générer n'importe quel format basé sur du texte (HTML, XML, CSV, ...). Le type de template le plus familier est un template *PHP* - un fichier texte analysé par PHP qui contient un mélange de texte et de code PHP.

Mais Symfony contient un langage de templates encore plus puissant appelé Twig. Twig vous permet d'écrire des templates lisibles et concis, qui plairont aux web designers, et qui sont, de plusieurs façons, plus puissants que les templates PHP.

Twig est un moteur de templates moderne pour PHP

- **Rapide** : Twig *compile* les templates en code PHP optimisé. La charge de travail comparé au code PHP a été réduite au strict minimum.
- **Sécurisé** : Twig possède un mode *sandbox* pour évaluer le code de template non approuvé. Cela permet à Twig d'être utilisé comme un langage de template pour applications ou les utilisateurs modifieraient le design du template.
- **Flexible**: Twig est alimenté par un flexible *lexer* et *parser*. Cela permet au développeur de définir ses tags et ses filtres personnalisés, et de créer ses propres DSL.

Prérequis

Twig nécessite au minimum **PHP 7.0.0** pour fonctionner.

Syntaxe spéciale:

Twig définit trois types de syntaxe spéciale:

`{{ ... }}`

"Dit quelque chose" : imprime une variable ou le résultat d'une expression vers le template.

`{% ... %}`

"Fait quelque chose" : un **tag** qui contrôle la logique du template; il est utilisé pour exécuter des instructions, des boucles par exemple.

`{# ... #}`

"Commente quelque chose": c'est l'équivalent de la syntaxe PHP `/* comment */`. Elle permet d'ajouter des commentaires d'une ou de plusieurs lignes. Le contenu des commentaires n'apparaîtra pas dans les pages générées.

Filtres :

Voir : http://twig.sensiolabs.org/doc/filters/index.html

Twig contient aussi des **filtres**, qui modifient le contenu avant qu'il soit généré. L'exemple de filtre ci-dessous permet de rendre la variable `title` entièrement en majuscule avant de la générer :

{{ title|upper }}

Twig est accompagné d'une longue liste de tags, filtres et fonctions qui sont disponibles par défaut. Vous pouvez même ajouter votre propre fonction, votre propre filtre *personnalisé* (et plus) via une *Twig Extension*.

Inclure d'autres Templates

Souvent, vous souhaiterez inclure le même template ou le même fragment de code sur plusieurs pages. Par exemple, dans une application avec des "articles d'actualités", le code template affichant un article pourrait être utilisé sur la page détaillant l'article, sur une page affichant les articles les plus populaires, ou sur la liste des derniers articles.

Inclure un template provenant d'un autre template est simple :

```
{{ include('article/article_details.html.twig', { 'article': article }) }}
```

Le template est inclus en utilisant la fonction `{{ include() }}` .

Liaisons vers les autres pages

Créer des liens vers les autres pages de votre application est l'une des tâches les plus communes d'un template. Au lieu de coder outre-mesure les URLs dans les templates, utilisez la fonction `path` de Twig pour générer des URLs basés sur la configuration de routage.

```
<a href="{{ path('welcome') }}">Home</a>
```

Vous pouvez aussi générer une URL absolue en utilisant la fonction `url()` de Twig :

```
<a href="{{ url('welcome') }}">Home</a>
```

Liaisons vers les assets

Les templates se réfèrent également aux images, à JavaScript, aux feuilles de style et aux autres assets. Évidemment, vous pouvez écrire le chemin de chaque asset (e.g. `/images/logo.png`), mais Symfony fournit une option plus dynamique via la fonction`asset()` de Twig.

Pour utiliser cette fonction, installez le package *asset* :

```
composer require asset
```

Vous pouvez maintenant utiliser la fonction `asset()` :

```
<link href="{{ asset('css/blog.css') }}" rel="stylesheet" />
```

Le but principal de la fonction `asset()` est de rendre votre application plus portable. Si votre application est à la racine de votre domaine (e.g. `http://example.com`), l'emplacement du rendu doit être `/images/logo.png`. Mais si votre application est dans un sous-répertoire (e.g. `http://example.com/my_app`), chaque emplacement doit être généré avec le sous-répertoire (e.g. `/my_app/images/logo.png`). La fonction `asset()` s'occupe de cela en générant les emplacements selon comment l'application est utilisée.

Comment vider les informations de débogage sur les templates Twig

Voir : https://symfony.com/doc/current/templating/debug.html

Quand vous utilisez des templates PHP, vous pouvez utiliser la fonction dump() de VarDumper si vous avez besoin de trouver rapidement la valeur d'une variable.

Premièrement, assurez vous qu'il est installé :

```
composer require var-dumper
```

ensuite dans vos fichiers templates :

```
{{ dump(your_variable) }}
```

Pensées finales

Le système de templating n'est qu'*un* des nombreux outils de Symfony. Et son travail est simple : nous permettre de générer un fichier de sortie HTML dynamique et complexe pouvant être retourné à l'utilisateur, envoyé dans un e-mail ou autres.

Service Container

Votre application est *remplie* d'objets utiles : un "Mailer" pourrait vous aider à envoyer des emails tandis qu'un autre objet pourrait vous aider à sauvegarder des choses dans la base de données. Presque *tout* ce que votre application "fait" est en réalité fait par un de ces objets. A chaque fois que vous installez un nouveau paquet, vous aurez accès à plus !

Dans Symfony, ces objets utiles sont appelés **services** et chaque service vit à l'intérieur d'un objet très spécial appelé **conteneur de services**. Le conteneur vous permet de centraliser le mode de construction des objets. Il vous facilite la vie, apporte une architecture forte et est super rapide !

Voir : https://symfony.com/doc/current/service_container.html

Quels sont les services disponibles? Découvrezle en lançant :

```
php bin/console debug:autowiring
```

L'option Autowire

Le fichier `services.yaml` à `autowire: true` dans la section `_defaults`. Cela s'applique donc à tous les services définis dans ce fichier. Avec ce paramètre, vous êtes capable de saisir des arguments dans la méthode `__construct()`, de vos services et le conteneur choisira automatiquement les arguments corrects. Cette entrée à été entièrement écrite autour de l'autowiring.

L'Autowiring vous permet de gérer les services dans le conteneur avec une configuration minimale. Il lit la saisie et relie chaque méthode au service correspondant. L'autowiring de Symfony est créé pour être prédictible: S'il n'est pas certain quelle est la dépendance qui doit être passée, vous verrez une exception .

Grâce à la section `_defaults` dans `services.yaml`, chaque service défini dans ce fichier `public: false` défaut.

Qu'est-ce que cela signifie? Quand un service **est** public, vous pouvez y accéder directement depuis l'objet container, qui est acces e depuis n'importe quel contrôleur qui hérite du`Controller`.

Comme un exercice pratique, vous devrez créer des services *privés* , qui aura lieu automatiquement. Et aussi, vous *ne* devrez *pas* utiliser la mét `$container->get()` pour récupérer/ aller chercher les services publiques. Mais, si vous *avez besoin* d'un service publique, pour annuler/passer outre /emporter sur les paramètres/réglages `public`.

Injections de service

Et si vous deviez accéder au service de connection : `logger` à partir de `MessageGenerator`? Pas de problème ! Créer une méthode `__construct()`avec un argument `$logger`qui a l'indication `LoggerInterface` . Mettez cela sur une nouvelle propriété `$logger`que vous utiliserez plus tard.

Et voilà ! Le container *automatiquement* passer le service `logger` quand il instancie le `MessageGenerator`. Comment sait-il comment faire cela? Autowiring.

La clé est l'indication `LoggerInterface` dans votre méthode `__construct()` et la config(uration) `autowire: true` en`services.yaml`. Quand tu indiques un argument, le container trouvera automatiquement le service correspondant. S'il n'y parvient pas, vous verrez clairement une exception avec un message d'aide.

pour info, cette méthode consistant à ajouter des dépendances à votre `__construct()` est appelé *dependency injection*. C'est une appellation/un terme complexe pour un concept simple.

Doctrine

Ce cours est un résumé depuis : http://symfony.com/doc/current/doctrine.html

Symfony ne fournie pas de composants spécifiques pour travailler avec la base de données, mais fournie plutôt une intégration avec Doctrine.

Cet article se concentre sur l'utilisation de doctrine ORM. Si vous préférez utiliser une version simplifiée pour interroger votre base de données, voyez _Doctrine DBAL_" à la place.

Installer Doctrine

Premièrement , nous allons installer Doctrine, avec le MakerBundle :

```
composer require doctrine maker
```

Configurer la base de données¶

Les informations de connexion à la base de données, sont stocker dans variable d'environnement appelé `DATABASE_URL`. En mode développement, vous pouvez personnaliser ces variables dans le fichier `.env`:

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"
```

Maintenant que vos paramètres de connexion ont été configurer Doctrine peut créer `db_name` pour vous:

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

Créer une Entité¶

Supposons que nous sommes en train de construire une application où des produits ont besoin d'être affichés, sans même réfléchir a propos de Doctrine ou des bases de données, nous savons que nous aurons besoins d'un objet `Produit` pour représenter nos produits Utilisez la commande `make:entity` pour créer la classe:

```
php bin/console make:entity Product
```

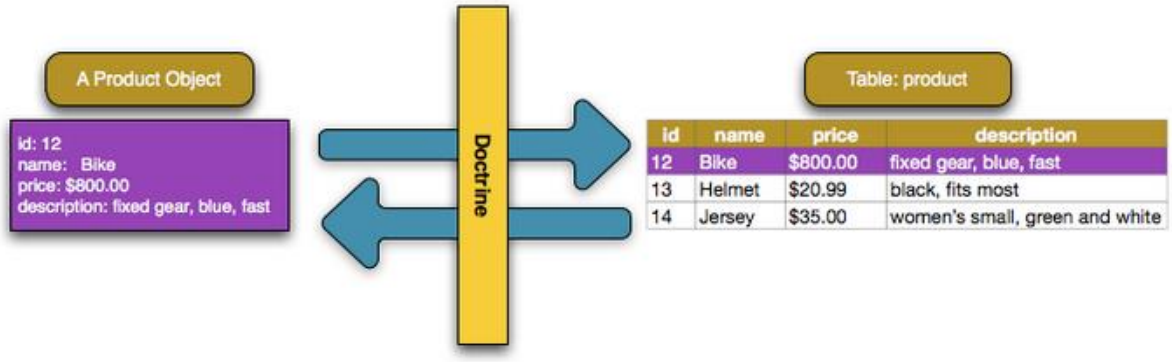
Vous avez maintenant un nouveau fichier `src/Entity/Product.php` .

Cette classe s'appelle une "entité". Et bientôt vous verrez comment sauvegarder et récupérer vos objets, depuis la table `produit` de votre base de donnée.

Associer d'autres Champs

Chaque propriété de notre entité `Produit` peut être associer avec une colonne de la table `produit` .

En ajoutant quelques configurations, Doctrine sera capable de sauvegarder un objet produit dans la table `produit` *et*interroger la table `produit` pour retourner des objets produits :



Migrations: Créer les tables de la base de données

Votre base de données ne contient pas de table`product` pour le moment. Pour ajouter cette table, nous pouvons utilisez le DoctrineMigrationsBundle, qui est déjà installé :

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

Si tout fonctionne correctement, vous devriez voir quelque chose comme:

Generated new migration class to "/path/to/project/doctrine/src/Migrations/Version20181122151511.php" from schema differences.

Si vous ouvrez ce fichier, vous verrez le code SQL pour mettre à jour votre base de données ! Pour exécuter ce code SQL, et vos migrations:

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

Cette commande exécute tous les fichiers de migration qui n'ont pas déjà été exécuté sur la base de données.

Persister vos Objects

Voir : http://symfony.com/doc/current/doctrine.html#creating-an-entity-class

Dans un controller, vous pouvez créer un objet `Product` , lui affecter des données et sauvegarder !

```
public function index()
```

```
{
    // you can fetch the EntityManager via $this->getDoctrine()
    // or you can add an argument to your action: index(EntityManagerInterface $entityManager)
    $entityManager = $this->getDoctrine()->getManager();

    $product = new Product();
    $product->setName('Keyboard');
    $product->setPrice(19.99);
    $product->setDescription('Ergonomic and stylish!');

    // tell Doctrine you want to (eventually) save the Product (no queries yet)
    $entityManager->persist($product);

    // actually executes the queries (i.e. the INSERT query)
    $entityManager->flush();

    return new Response('Saved new product with id '.$product->getId());
}
```

La Documentation complète ici : http://symfony.com/doc/current/doctrine.html