Symfony

Base de Données et Templating

Maxime Puys

22 mars 2025

Plan

Base de Données avec Doctrine

Introduction à Doctrine

- ORM recommandé des les SGDBR dans Symfony.
- Accès de bas niveau pour exécuter des requêtes SQL brutes (similaire au PDO de PHP).
- Compatible avec MySQL, PostgreSQL, SQLite, MongoDB, etc

Rappels ORM

- Sans requête préparée :

```php
$requete = "SELECT * FROM ma_table WHERE mon_champ = ma_valeur";
$resultats = $connexion->query($requete);
```

- Avec requête préparée :

```php
$requete = "SELECT mon_champ FROM ma_table WHERE mon_champ = :ma_valeur";
$resultats = $connexion->prepare($requete);
$resultats->execute([":ma_valeur" => 5]);
```

- Avec un ORM :

```php
$user = new User();
$user->setName('John');
$user->setPassword('Doe');
$user->save();
```

Rappels ORM

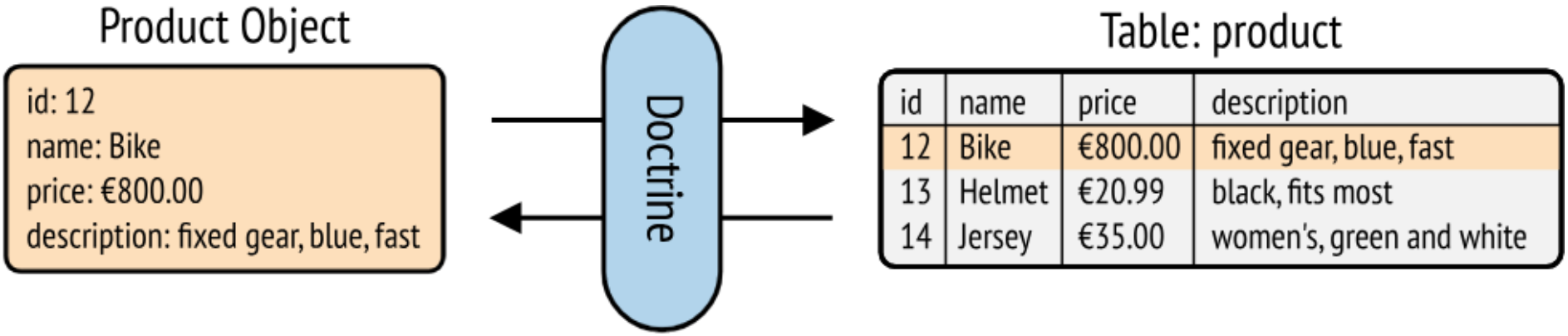


Figure – Doctrine

Installation de Doctrine

Installation de Doctrine via `composer` :

```
1 composer require symfony/orm-pack
2 composer require --dev symfony/maker-bundle
```

- `orm-pack` : Installation de Doctrine
- `maker-bundle` : Permet la génération de code

⇒ Généralement installé automatiquement.

Configuration de la Base de Données

- La connexion à la base de données est définie dans une variable d'environnement `DATABASE_URL`.
- Pour le développement, mettre dans `.env` :

```
1  // .env :
2  # Exemple avec MySQL
3  DATABASE_URL="mysql://user:passwd@127.0.0.1:3306/nom_base?serverVersion=5.7"

4  # Exemple avec SQLite
5  DATABASE_URL="sqlite:///%kernel.project_dir%/var/app.db"
```

Création d'une Entité

- Création d'une classe d'entité pour représenter les objets dans la BDD, par exemple, `Produit` :

```
$ symfony console make:entity

Nom de la classe :
> Produit

Nom de la propriété :
> nom
Type de champ :
> string
Longueur du champ :
> 255
```

Classe Entité Générée

- Voici à quoi ressemble la classe `Produit` générée :

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

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
class Produit
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    private $nom;
    // ...
}
```

Rappels ORM

⇒ Rappel : Une entité == Une table en BDD

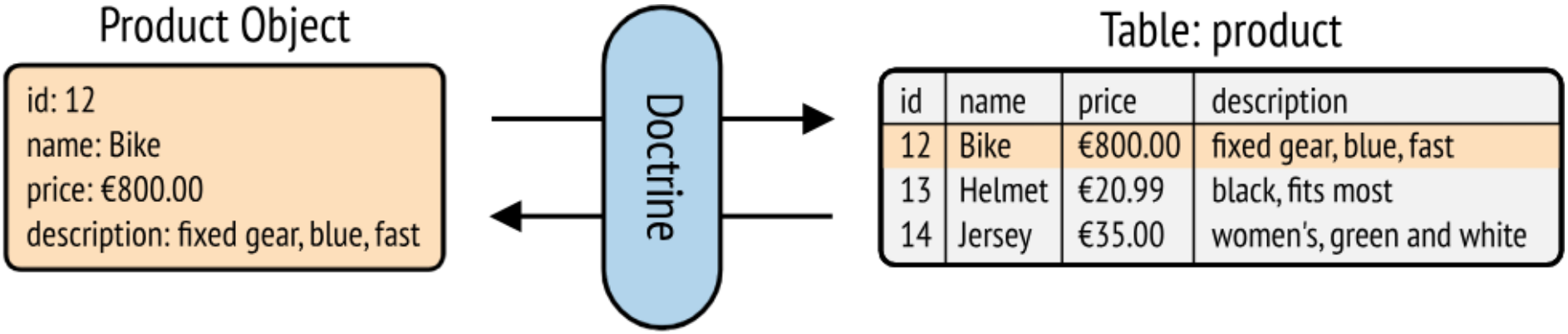
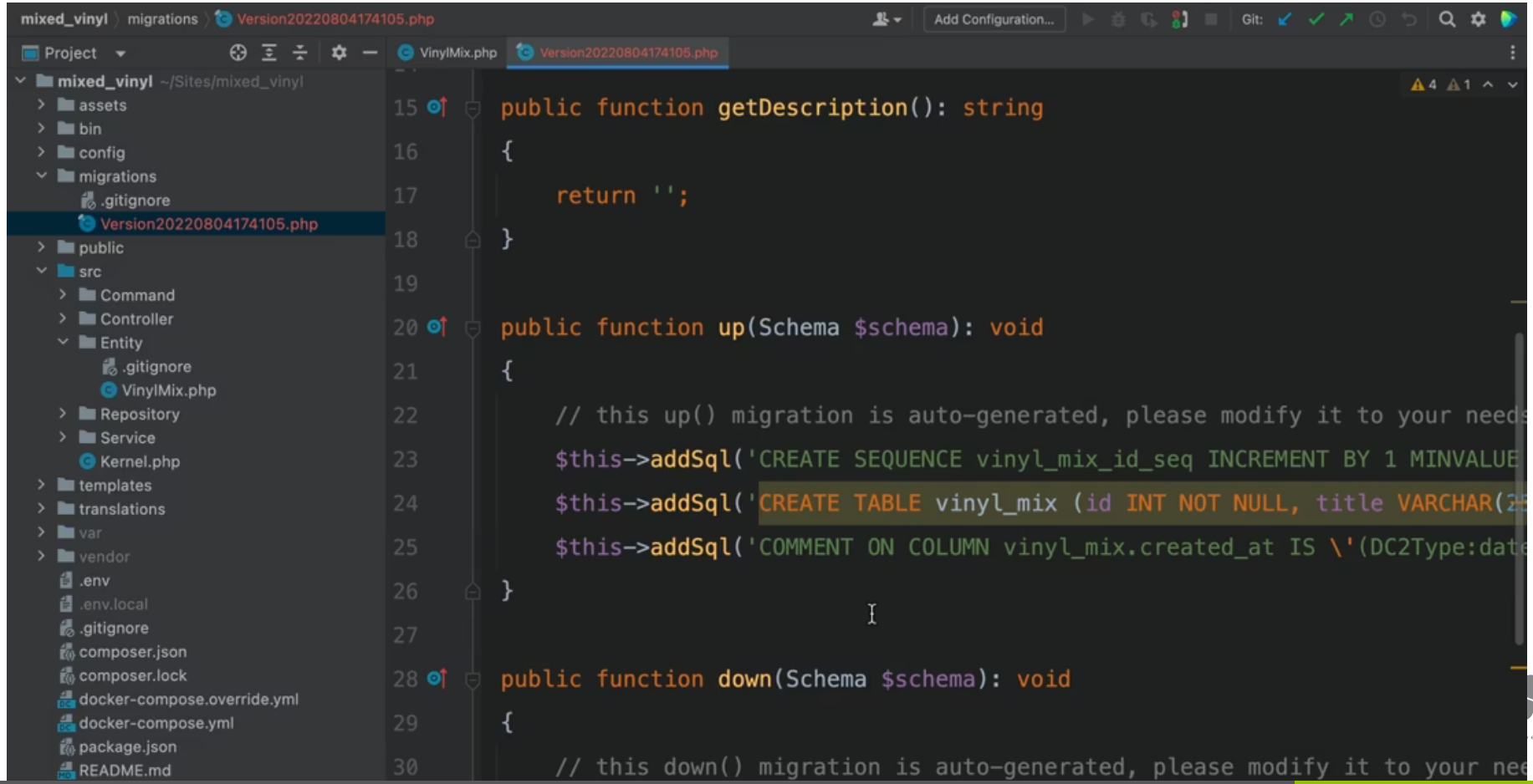


Figure – Doctrine

Migrations : Créer les Tables de Base de Données

Utilisez le `DoctrineMigrationsBundle` pour créer les tables :

```
1 symfony console make:migration
2 symfony console doctrine:migrations:migrate
```



Migrations : Créer les Tables de Base de Données

- Rollback du code ⇒ Facile avec GIT
- Rollback de la structure de la BDD ??
- Rollback de la structure des données déjà en BDD ?/

Migrations : Créer les Tables de Base de Données

- Rollback du code ⇒ Facile avec GIT
- Rollback de la structure de la BDD ??
- Rollback de la structure des données déjà en BDD ?/

- Les migration sont permettent du versionning dans la structure des tables !
- Elle contiennent le code permettant de modifier la strucure : `up`
- Et de revenir en arrière : `down`

Migrations : Créer les Tables de Base de Données

- Rollback du code ⇒ Facile avec GIT
- Rollback de la structure de la BDD ??
- Rollback de la structure des données déjà en BDD ?/

- Les migration sont permettent du versionning dans la structure des tables !
- Elle contiennent le code permettant de modifier la strucure : `up`
- Et de revenir en arrière : `down`

- Aussi utiles pour un travail collaboratif :
  - Si un collègue modifie la structure de la BDD, je peux appliquer la migration au lieu de faire la modif à la main !

L'Entity Manager de Doctrine

Requis pour la pluspart des manipulations de la BDD !

- Interface de Doctrine

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

use Doctrine\ORM\EntityManagerInterface;

class ProductController extends AbstractController
{
    public function __construct(EntityManagerInterface $entityManager) {}

    #[Route('/product', name: 'create_product')]
    public function createProduct(EntityManagerInterface $entityManager): Response
}
```

Persiste des Objets dans la Base de Données

- Dans le code, créez un nouvel objet, définissez ses données et sauvegardez-le :

```php
// src/Controller/ProduitController.php
// ...
$produit = new Produit();
$produit->setNom('Clavier');
// ...
$entityManager->persist($produit); //A utiliser qu'en cas de nouvelle entité!
$entityManager->flush();
```

- Rappel : Injection de dépendances pour accéder à `$entityManager`
- Penser à appeller `flush` pour executer les requêtes

Validation des Objets

Symfony réutilise les métadonnées Doctrine pour la validation :

```php
// src/Controller/ProduitController.php
public function createProduct(ValidatorInterface $validator): Response {
    $product = new Product();
    // ...
    $erreurs = $validator->validate($produit);
    if (count($erreurs) > 0) {
        // Gérer les erreurs
        return new Response((string) $errors, 400);
    }
}
```

Récupération d'Objets de la Base de Données

- A la création de l'entité, création automatique d'un `Repository`

```php
// src/Repository/ProductRepository.php
namespace App\Repository;

use App\Entity\Product;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Product::class);
    }
}
```

Méthodes du Repository

Le repository offre plusieurs méthodes utiles :

- `find($id)` pour chercher par clé primaire.
- `findOneBy(['name' => 'Nom'])` pour chercher par critères.
- `findBy(['name' => 'Nom'], ['price' => 'ASC'])` pour plusieurs objets.
- `findAll()` pour tous les objets.

Méthodes Personnalisées

Ajoutez des méthodes personnalisées pour des requêtes complexes dans votre `ProductRepository`.

```php
// src/Repository/ProductRepository.php
class ProductRepository extends ServiceEntityRepository {
    // ...
    public function findAllGreaterThanPrice(int $price): array {
        $entityManager = $this->getEntityManager();
        $query = $entityManager->createQuery(
            'SELECT p
            FROM App\Entity\Product p
            WHERE p.price > :price
            ORDER BY p.price ASC'
            // ATTENTION, ceci n'est pas du pure SQL !!
        )->setParameter('price', $price);

        // returns an array of Product objects
        return $query->getResult();
    }
}
```

Utilisation du Repository dans le Controller

- Récupération manuelle du `Repository` via l'`EntityManagerInterface` :

```php
// src/Controller/ProductController.php
public function show(EntityManagerInterface $entityManager, int $id): Response {
    // Récupération du repository de produit
    $productRepo = $entityManager->getRepository(Product::class);

    // Récupération du produit via le repository
    $product = $productRepo->find($id);

    // Utilisation des données
    if (!$product) {
        throw $this->createNotFoundException(
            'No product found for id '.$id
        );
    }

    return new Response('Check out this great product: '.$product->getName());
}
```

Utilisation du ProductRepository

- Sinon injection automatique de `ProductRepository` via l'autowiring :

```php
// src/Controller/ProductController.php
class ProductController extends AbstractController
{
    public function show(ProductRepository $productRepository, int $id): Response
    {
        $product = $productRepository->find($id);
        // ...
    }
}
```

- Mais ne permet pas le `flush` sans l'`EntityManagerInterface`

Mise à Jour d'un Objet

```php
// src/Controller/ProductController.php
public function update(EntityManagerInterface $entityManager, int $id): Response {
    $product = $entityManager->getRepository(Product::class)->find($id);

    if (!$product) {
        // ...
    }

    $product->setName('New product name!');
    $entityManager->flush();

    // ...
}
```

Suppression d'un Objet

Supprimer un objet en utilisant la méthode `remove()` :

```
1 $entityManager->remove($product);
2 $entityManager->flush();
```

Utilisation Directe de SQL

- Très similaire à PDO :

```php
$conn = $entityManager->getConnection();

$sql = '
    SELECT * FROM product p
    WHERE p.price > :price
    ORDER BY p.price ASC
    ';
$stmt = $conn->prepare($sql);
$resultSet = $stmt->executeQuery(['price' => $price]);

// returns an array of arrays (i.e. a raw data set)
return $resultSet->fetchAllAssociative();
```

- Attention ! On ne récupère pas des objets mais des données brutes (ex : un dictionnaire)

Bonne pratique : Utilisation du QueryBuilder

- Doctrine propose le QueryBuilder permettant une construction dynamique des requêtes SQL

```php
// src/Repository/ProductRepository.php
class ProductRepository extends ServiceEntityRepository {
    // ...
    public function findAllGreaterThanPrice(int $price): array {
        $query = $this->createQueryBuilder()
        ->select('p') //Select all entity, return Product object
        ->where('p.price > :price')
        ->orderBy('p.price', 'ASC')
        ->setParameter('price', $price);

        // returns an array of Product objects
        return $query->getResult();
    }
```

Bonne pratique : Utilisation du QueryBuilder (suite)

```
1  public function getHigherPriceAndAfterChristmas(int $price): array {
2      $qb = $this->createQueryBuilder();
3      $qb
4      ->select('p') //Select all entity, return Product object
5      ->where('p.price > :price')
6      ->andWhere(
7          $qb->expr()->orX(
8              $qb->expr()->eq('p.id', '667'), //if id = 667...for some reasons
9              $qb->expr()->gt('p.last_edit', ':lastEdit') //greather than lastEdi
0          )
1      )
2      ->orderBy('p.price', 'ASC')
3      ->setParameters([
4              'price' => $price,
5              'lastEdit' => '2024-12-25'
6      ]);
7      return $query->getResult();
8  }
9  }
```

Templating avec Twig

Introduction au Langage de Templating Twig

- Twig : moteur de template pour PHP
- Va utiliser des variables PHP pour générer du contenu statique
- Peut-être utilisé dans des codes HTML, CSS, JS, etc
- Twig utilise trois constructions principales :
  - `{{ ... }}` pour afficher le contenu d'une variable ou le résultat d'une expression.
  - `{% ... %}` pour exécuter de la logique, comme une condition ou une boucle.
  - `{# ... #}` pour ajouter des commentaires (non inclus dans la page rendue).

Exemple de Template Twig

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Welcome to Symfony!</title>
5      </head>
6      <body>
7          <h1>{{ page_title }}</h1>

8          {% if user.isLoggedIn %}
9              Hello {{ user.name }}!
0          {% endif %}
1      </body>
2  </html>
```

Création de Templates

Créez un fichier dans le répertoire `templates/` :

```twig
{# templates/user/notifications.html.twig #}
<h1>Hello {{ user_first_name }}!</h1>
<p>You have {{ notifications|length }} new notifications.</p>
```

Puis, créez un contrôleur pour rendre ce template :

```php
// src/Controller/UserController.php
class UserController extends AbstractController
{
    public function notifications(): Response
    {
        $userFirstName = '...';
        $userNotifications = ['...', '...'];
        return $this->render('user/notifications.html.twig', [
            'user_first_name' => $userFirstName,
            'notifications' => $userNotifications,
        ]);
    }
}
```

Emplacement des Templates

- Les templates sont stockés par défaut dans le répertoire `<votre_projet>/templates/`
- `render("product/index.html.twig")` dans le code ⇒ `<votre_projet>/templates/products/index.html.twig` sur le disque
- / !\ Penser à l'extension `.twig` / !\
- Si `{{ foo.bar }}` dans le template, Twig va tenter de résoudre le symbole comme :
  1. `$foo['bar']`
  2. `$foo->bar`
  3. `$foo->bar()`
  4. `$foo->getBar()`
  5. etc.
  6. `null` or erreur suivant la configuration

Lien vers les Pages / CS, JS, etc

- Utilisez la fonction `path()` pour générer des URL basées sur la configuration du routage.

```
#[Route('/article/{slug}', name: 'blog_post')]
public function show(string $slug): Response {}
```

```
<a href="{{ path('blog_post', {slug: post.slug}) }}">{{ post.title }}</a>
```

- Utilisez la fonction `asset()` pour générer l'URL des assets statiques.

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!"/>
<img src="{{ absolute_url(asset('images/logo.png')) }}" alt="Symfony!"/>
```

- `https://example.com` $\Rightarrow$ `/images/logo.png`
- `https://example.com/my_app` $\Rightarrow$ `/my_app/images/logo.png`

Variable Globale App

- Symfony injecte automatiquement une variable `app` dans chaque template Twig, offrant un accès à des informations de l'application :

```
1 <p>Username: {{ app.user.username ?? 'Utilisateur anonyme' }}</p>
2 {% if app.debug %}
3     <p>Méthode de la requête : {{ app.request.method }}</p>
4     <p>Environnement de l'application : {{ app.environment }}</p>
5 {% endif %}
```

Rendu de Templates dans les Services

- Injectez Twig dans vos propres services pour utiliser sa méthode `render()` :

```php
// src/Service/SomeService.php
class SomeService
{
    private $twig;

    public function __construct(Environment $twig) {}

    public function someMethod()
    {
        $htmlContents = $this->twig->render('product/index.html.twig', [
            // ...
        ]);
    }
}
```

Render vs. renderView

- `renderView()` renvoie uniquement le contenu créé par le template.

```php
// src/Controller/ProductController.php
class ProductController extends AbstractController
{
    public function index(): Response
    {
        // the `render()` method returns a `Response` object with the
        // contents created by the template
        return $this->render('product/index.html.twig', [
            'category' => '...',
            'promotions' => ['...', '...'],
        ]);

        // the `renderView()` method only returns the contents created by the
        // template, so you can use those contents later in a `Response` object
        $contents = $this->renderView('product/index.html.twig', [
            'category' => '...',
            'promotions' => ['...', '...'],
        ]);
        return new Response($contents);
    }
}
```

Rendu Direct de Templates depuis une Route

- Rendez des pages statiques directement depuis la définition de la route avec `TemplateController` :

```yaml
# config/routes.yaml
acme_privacy:
    path:           /privacy
    controller:     Symfony\Bundle\FrameworkBundle\Controller\TemplateController
    defaults:
        # the path of the template to render
        template:  'static/privacy.html.twig'
```

Définition de Variables

- Vous pouvez attribuer des valeurs aux variables avec le tag `set` :

```
1 {% set foo = 'foo' %}
2 {% set foo = [1, 2] %}
3 {% set foo = {'foo': 'bar'} %}
```

Filtres

- Les variables peuvent être modifiées par des filtres, séparés par un symbole pipe (|). Les filtres peuvent être chaînés :

```
{{ nom|striptags|title }}
```

- Pour appliquer un filtre à une section, utilisez le tag `apply` :

```
{% apply upper %}
    Ce texte devient en majuscules
{% endapply %}
```

Fonctions

- Les fonctions génèrent du contenu. Elles sont appelées par leur nom suivi de parenthèses :

```
{% for i in range(0, 3) %}
    {{ i }},
{% endfor %}
```

- Les arguments nominaux sont supportés dans les fonctions et filtres :

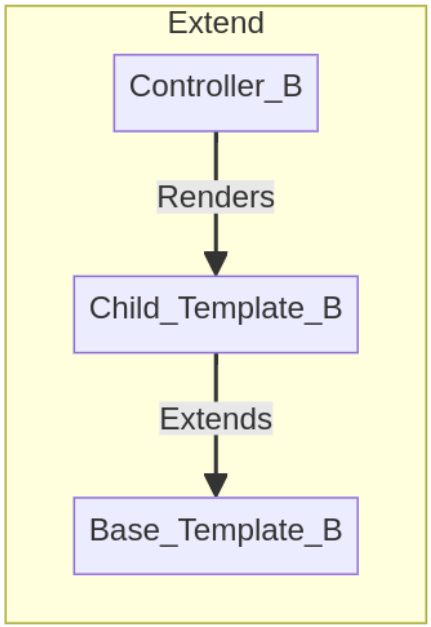
```
{% for i in range(low=1, high=10, step=2) %}
    {{ i }},
{% endfor %}
```

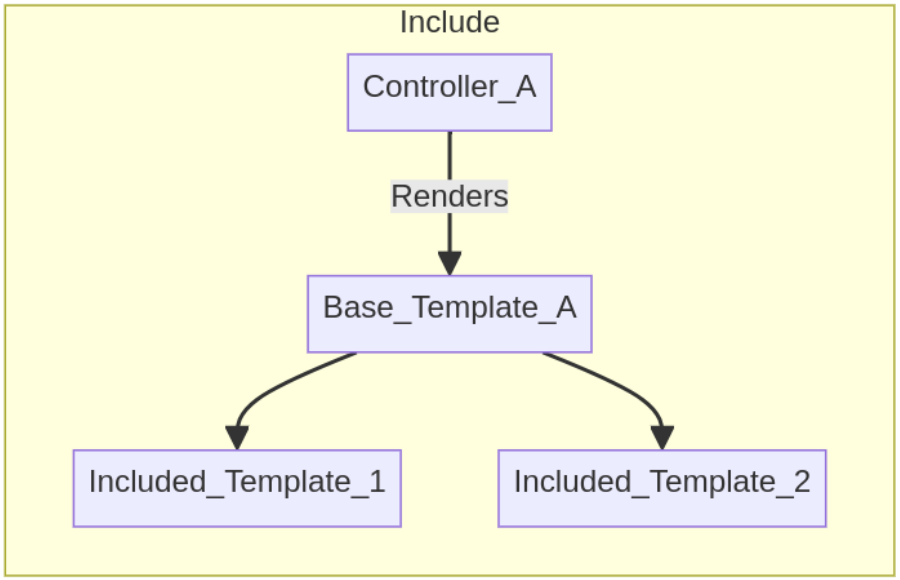
Structures de Contrôle

- Les structures de contrôle dirigent le flux du programme - conditionnels (`if/elseif/else`), boucles (`for`), etc.

```
{% if users|length > 0 %}
    <ul>
        {% for user in users %}
            <li>{{ user.username|e }}</li>
        {% endfor %}
    </ul>
{% endif %}
```

Inclusion vs. Heritage





Inclusion d'Autres Templates

- La fonction `include` permet d'inclure un template :

```
{{ include('sidebar.html.twig') }}
```

- Les templates inclus ont le même contexte que le template appelant.

```
{% for box in boxes %}
    {{ include('render_box.html.twig') }}
{% endfor %}
```

- Variable `box` différente à chaque inclusion de `render_box.html.twig`

Héritage de Template

- Héritage : squelette de base et blocs définis dans les templates enfants

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <title>{% block title %}{% endblock %} - My Webpage</title>
        {% endblock %}
    </head>
    ...
</html>
```

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

{% block titre %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
```

Échappement HTML

- Twig peut échapper automatiquement le HTML.

```
{% autoescape %}
    Everything will be automatically escaped in this block (using the HTML strategy
{% endautoescape %}
```

- Sinon escape manuel avec le filtre `escape` :

```
{{ utilisateur.nom|escape('html') }}
```

Macros et Expressions

- Les macros sont des racourcis sous forme de fonction :

```twig
{# forms.html.twig #}
{% macro input(name, value, type = "text") %}
    <input type="{{ type }}" name="{{ name }}" value="{{ value }}" />
{% endmacro %}
```

```twig
{# ma-page.html.twig #}
{% include('forms.html.twig') %}
<p>{{ forms.input('username') }}</p>
<p>{{ forms.input('password', null, 'password') }}</p>
```

Expressions dans Twig - Littéraux

- **Attention**, les oppérations déportent la logique du controlleur/modèle dans la vue !
  - Risque de se mélanger !
- Les formes les plus simples d'expressions sont les littéraux :
  - Valeurs en dur, à utiliser dans des expressions avec des variables
- Exemples :
  - Chaînes : `"Bonjour le monde"`, `'C\'est bon'`
  - Nombres : `42`, `42.23`
  - Tableaux : `["foo", "bar"]`
  - Dictionnaires : `{"foo": "bar"}`, `{foo: 'foo', bar: 'bar'}`
  - Booléens : `true`, `false`
  - Null : `null`

Expressions dans Twig - Mathématiques

- Twig autorise les opérations mathématiques dans les templates.

```
{# `+` : Addition #}
{{ date + 7 }}

{# `-` : Soustraction #}
{{ heure - 2 }}

{# `/` : Division #}
{{ set moyenne = somme / longueur }}

{# `%` : Modulo #}
{{ (nombre % 2) == 0 }}

{# `//` : Division entière #}
{{ set nbGroups = nbEleves // 14 }}

{# `*` : Multiplication #}
{{ quantité * prixUnitaire }}

{# `**` : Puissance #}
{{ set surface = longueur ** 2 }}
```

Expressions dans Twig - Logique

- Les expressions peuvent être combinées avec des opérateurs logiques :
  - `and` : Retourne vrai si les deux opérandes sont vrais.
  - `or` : Retourne vrai si l'un des opérandes est vrai.
  - `not` : Négation d'une déclaration.
  - `(expr)` : Groupe une expression.
- Twig supporte les opérateurs de comparaison : `==`, `!=`, `<`, `>`, `>=`, `<=`.

Expressions dans Twig - Autres Opérateurs

```
1 {% if 'Fabien' starts with 'F' %}{% endif %}
2 {% if phone matches '/^[\\d\\.]+$/' %}{% endif %}
3 {{ "Hello #{name}, how are you?" }}
```

Opérateurs spéciaux :

- `1..10` : Crée une séquence.
- `"Hello " ~ "World!"` : Concatène des chaînes.
- `object.attribute`, `array[index]` : Accède à un attribut d'une variable.
- `condition ? then : else` : Opérateur ternaire.
- `age ?? 0` : Coalescence nulle (retourne la valeur si existe sinon).
- `post.status is "PUBLISHED"` : Egalité
- `groupeTP in 1..10` : Inclusion
- `not` : Inverse