



Framework Symphony



Présenté par : Sana EL ATCHIA



Introduction(1/2)

- **Symfony est un ensemble de composants PHP réutilisable.**
=> Un composant est une librairie qui a été conçu pour répondre à un besoin précis.
- **Symfony est un Framework PHP pour des applications web**
=> un ensemble d'outils qui nous permet de faciliter la conception de notre application web
- **Un Framework est Cadre De Travail (Boite à outils)**

Introduction(2/2)

- Symfony : Framework côté serveur basé sur PHP
- **Intérêts :**
 - structuration du code (**MVC**)
 - simplification du développement
 - nombreux modules existants (bibliothèques)
- **Points saillants :**
 - routage facile et url propres (via **annotations/Attributs**)
 - contrôleurs : PHP et objet
 - manipulation des bases de données : **Doctrine**
 - langage de Template : **Twig**
 - gestion de formulaires facilitée

Plan

1. Démarrage et organisation
2. Template
3. Contrôleurs
4. Base de données
5. Formulaires

Démarrage et organisation

1. Installation de Symfony
2. Méthode d'organisation de code
 - Architecture MVC
 - Organisation des fichiers

Mise en place de Symfony–Windows (1/2)

1. Installation PHP 8

- PHP 8.2 doit déjà être installé sur l'environnement de travail.

2. Installation composer

- Composer est un **gestionnaire de dépendances** PHP. Il permet d'installer des librairies externes sur un projet PHP.
- Voici le lien d'installation : <https://getcomposer.org>

3. Installation Scoop

- Scoop est un installeur en ligne de commande pour Windows.
- Voici le lien d'installation : <https://scoop.sh/>
- Dans le terminal **Power Shell**, exécutez les commandes suivantes :
 - **Set-ExecutionPolicy** RemoteSigned -Scope CurrentUser
 - **irm** get.scoop.sh | **iex**

Mise en place de Symfony–Windows (2/2)

3. Installation Symfony CLI

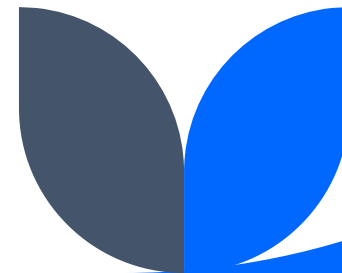
- Symfony CLI permet de lancer un serveur PHP local afin de faire tourner votre application
- Voici le lien d'installation : <https://symfony.com/download>
- pour l'installer, il suffit de taper la commande suivante (cmd/admin ou power shell) :
 - **scoop install symfony-cli**
 - **scoop update symfony-cli**

4. Installation de git

- Voici le lien d'installation : <https://git-scm.com/download/>
- pour installer GIT, il suffit de taper la commande suivante :
 - **winget install --id Git.Git -e --source winget**
- Redémarrer la machine

Initialisation Symfony

- Afficher la liste des différentes versions PHP en local : `symfony local:php:list`
- Créer une application Web utilisant Symfony :
 - Méthode 1 : `symfony new lgc_app --version="7.0.*" --webapp`
 - Méthode 2 : `symfony new lgc_app --full` ou `symfony new lgc_app --version="6.*.*" --full`
- Afficher la version actuelle de votre projet Symfony : `symfony console about`
- Démarrer un serveur local : `symfony server:start` ou `symfony -d server:start`
- Arrêter un serveur local : `symfony server:stop`
- Afin d'assurer une navigation locale sécurisée, c-à-d parcourir votre application locale avec HTTPS au lieu de HTTP, installez l'autorité de certification locale (local certificate authority) avec la commande : `symfony server:ca:install`



Mettre l'application sur GitHub

Ajouter un dépôt distant

- Créer un repository distant sur github : app_produit

Associer mon dépôt local (application symfony) à mon dépôt distant :

- `cd app_produit`
- `git init`
- `git remote add origin https://github.com/Sana-El-Ath/app_produit.git`

Installer un composant Symfony

- Voici le lien pour installer les composants Symfony : <https://symfony.com/components>
- Pour installer un composant Symfony sur un projet PHP, on utilise la syntaxe suivante :

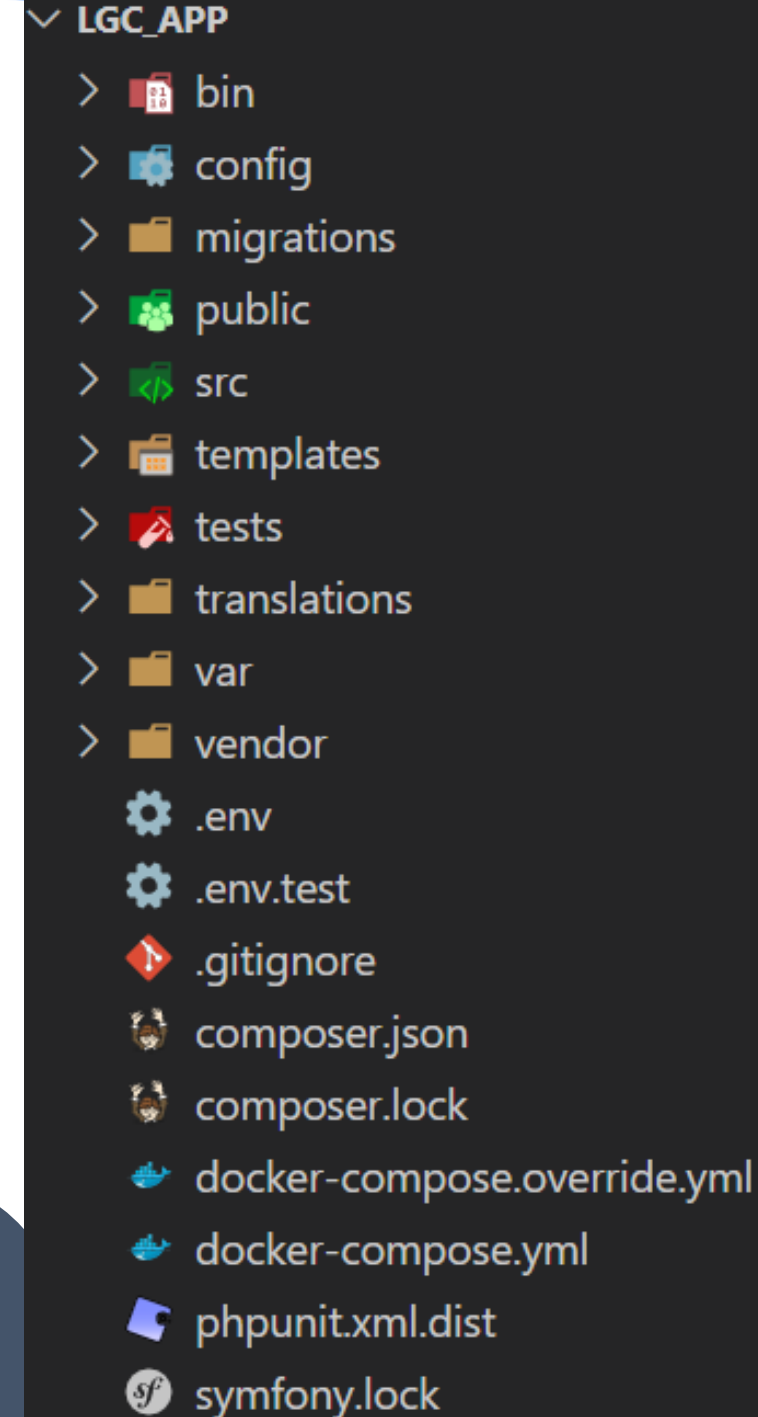
composer require nomComposant/Dépendance

- `composer require symfony/maker-bundle --dev`
 - `composer require symfony/orm-pack`
 - `composer require symfony/security-bundle`
 - `composer require symfony/form`
 - `composer require symfony/validator`
 - `composer require symfony/phpunit-bridge`
 - `composer require --dev orm-fixtures`
 - `composer require fakerphp/faker`
- Consulter les différents composants installés, dans les fichiers suivants :
 - **Vendor** : ce dossier contient les packages d'installation de chaque dépendance
 - **Composer.json** : C'est le fichier de configuration de composer, il contient les différentes dépendances installées
 - **Composer.lock** : ce fichier mentionne les différentes versions de dépendances utilisées, On ne peut pas le modifier

Structure du framework

- **Bin** : Ce dossier contient les exécutables disponibles dans le projet, que ce soit ceux fournis avec le Framework (la console Symfony) ou ceux des dépendances (phpunit,...).
- **Config** : contient toute la configuration de votre application, que ce soit le framework, les dépendances (Doctrine, Twig, Monolog) ou encore les routes.
- **migrations** : migration du projet à chaque changement effectuer sur la base de données à l'aide de l'ORM Doctrine
- **Public** : ne contient que le contrôleur frontal de l'application. Il reçoit toutes les requêtes des utilisateurs.
- **Src** : contient l'application (Contrôleurs, formulaires, écouteurs d'événements, modèles, le moteur Kernel).
- **Templates** : contient les gabarits (layouts) qui sont utilisés dans l'application.
- **Tests** : contient les tests unitaires, d'intégration et d'interfaces.

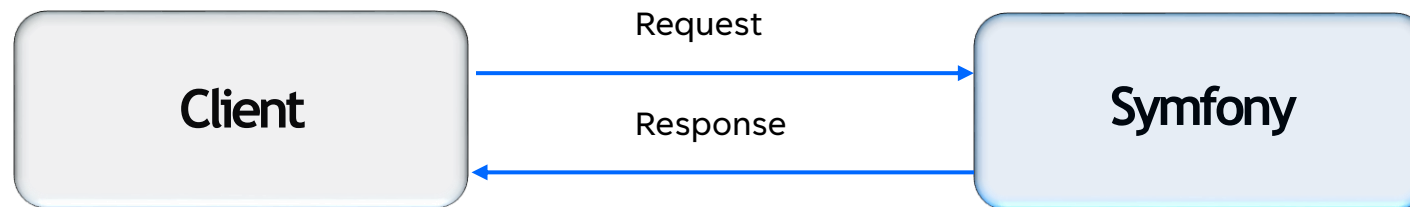
RMQ : Par défaut, l'espace de nom du dossier **tests** est **App\Tests** et celui du dossier **src** est **App**



Fonctionnement Symfony

Symfony basé sur l'échange HTTP Request/Response :

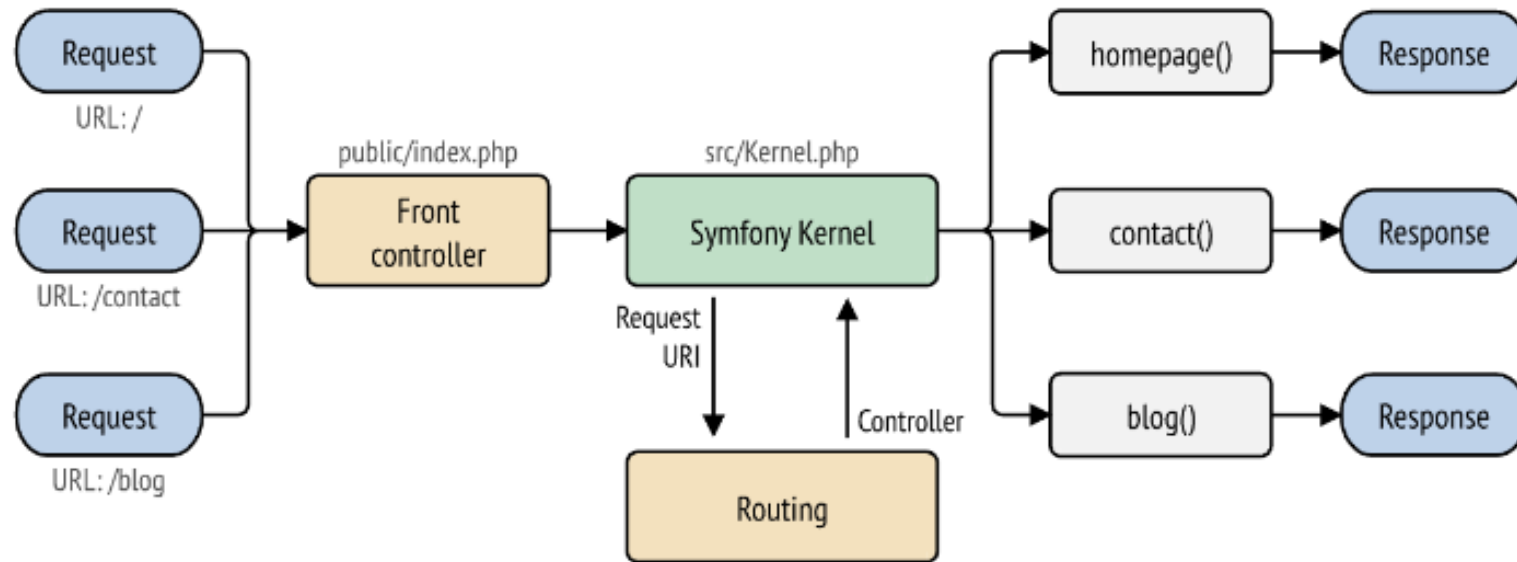
1. L'utilisateur applique une action sur le navigateur (clique sur un lien, envoie d'un formulaire HTML)
2. Le navigateur envoie une requête au Symfony Dans le serveur,
3. Symfony crée un objet **Request** contenant les données envoyées par l'utilisateur (les données du formulaire, les paramètres du URL,...)
4. Symfony génère un objet **Response** à partir des données envoyées avec l'objet Request
5. Symfony retourne une réponse au navigateur
6. Le navigateur afficher la réponse à l'utilisateur (page html, json,...)



Architecture MVC

Modèle-vue-contrôleur

- Symfony utilise l'architecture MVC pour structurer le code:



Source : https://symfony.com/doc/current/introduction/http_fundamentals.html#the-symfony-application-flow

- La partie serveur est coupée en trois morceaux :
 - le **modèle** qui implémente des services (p. ex. gestion de la base de données)
 - le **contrôleur** qui prend les décisions pour générer les pages demandées
 - la **vue (Template)** qui organise la présentation (affichage).

MVC en pratique - Contrôleur

- Contrôleurs en PHP (orienté objets) :

- **le kernel Symfony** fait le chef d'orchestre (front controller)

- selon l'url reçue, invoque un contrôleur (controller) pour générer la page demandée
- gestion des url par des attributs/des annotations

- **un contrôleur pour chaque « section » du site** (p. ex. login, enregistrement, chat, etc.)

- un contrôleur (classe PHP) : plusieurs actions, une action pour chaque page à générer

- **une action d'un contrôleur (fonction)** : implémente la logique de génération d'une page :

- récupération des données / de l'input, traitement et envoi aux vues



MVC en pratique – Vue en Twig

- une vue par « type de page »
- langage de Template simple et puissant
- décrit la présentation de la page en fonction de plusieurs paramètres passés par le contrôleur
- séparation logique de « calcul » de la page / affichage
- héritage

MVC en pratique - Modèle

- Un modèle est un ensemble de services (p. ex. gestion de la base de données, des formulaires, de la sécurité (authentification), des messages et mails, etc.)
 - la plupart des services courants fournis par Symfony
 - service de gestion de la base de données (mysql) : **Doctrine**
 - correspondance : données dans la bd \Leftrightarrow objets PHP
 - possibilité de créer ses propres services (accessibles par tous les contrôleurs)

Les fondamentaux de Symfony

1. **Routing** est un composant Symfony qui permet de lier une URL à une action PHP. Il supporte de nombreux formats de déclaration **les attributs**, **YAML** ou **les annotations**.
2. **Contrôleur (C)** : permet d'appeler les entités, si besoins et de charger les vues
➡ src/Controller/**Nom**Controller.php
3. **Entités (M)** : est utilisée pour communiquer avec la base de données
➡ src/Entity/**NomEntite**.php
4. **Vue (V)** : Permet d'afficher la page HTML à l'utilisateur
➡ templates/**nomVue**.html.twig

Exécution de commandes avec Symfony (1/2)

- Grace au fichier `bin/console.php` , on peut exécuter plusieurs commandes Symfony
- On exécute ce fichier avec l'interpréteur PHP



```
php bin/console

console > ...
#!/usr/bin/env php
<?php

use App\Kernel;
use Symfony\Bundle\FrameworkBundle\Console\Application;

if (!is_file(dirname(__DIR__).'/vendor/autoload_runtime.php')) {
    throw new LogicException('Symfony Runtime is missing. Try running "composer require symfony/runtime".');
}

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    $kernel = new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);

    return new Application($kernel);
};
```

Exécution de commandes avec Symfony (2/2)

❖ **Maker** : C'est un package qui permet d'installer les entités, les vues et les contrôleurs...

- La commande suivante permet de lister les commandes possibles sous l'espace de noms make:

php bin/console list make ⇔ symfony console list make

```
make
make:auth          Creates a Guard authenticator of different flavors
make:command        Creates a new console command class
make:controller     Creates a new controller class
make:crud           Creates CRUD for Doctrine entity class
make:docker:database Adds a database container to your docker-compose.yaml file
make:entity         Creates or updates a Doctrine entity class, and optionally an API Platform resource
make:fixtures       Creates a new class to load Doctrine fixtures
make:form           Creates a new form class
make:message        Creates a new message and handler
make:messenger-middleware Creates a new messenger middleware
make:migration       Creates a new migration based on database changes
make:registration-form Creates a new registration form system
make:reset-password Create controller, entity, and repositories for use with symfonycasts/reset-password-bundle
make:serializer:encoder Creates a new serializer encoder class
make:serializer:normalizer Creates a new serializer normalizer class
make:subscriber     Creates a new event subscriber class
make:test           [make:unit-test|make:functional-test] Creates a new test class
make:twig-extension Creates a new Twig extension class
make:user           Creates a new security user class
make:validator       Creates a new validator and constraint class
make:voter          Creates a new security voter class
```

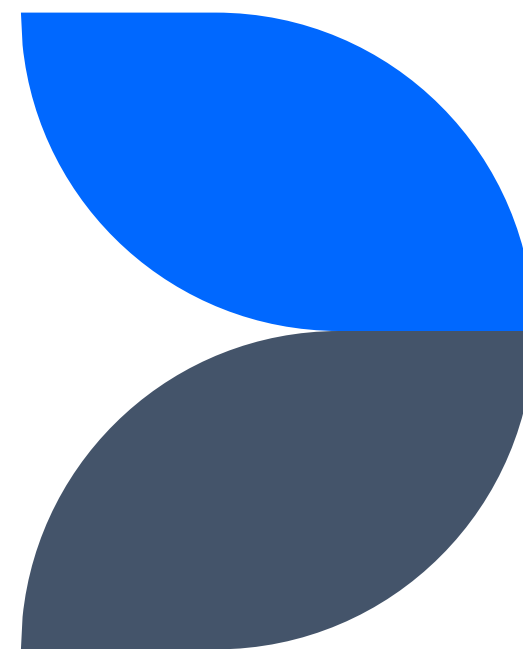
- La commande suivante permet de créer un contrôleur :

symfony console make:controller NomController ⇔ php bin/console make:controller NomController

- La commande suivante permet de créer une entité :

symfony console make:entity NomEntity ⇔ php bin/console make:entity NomEntity

Contrôleurs



Ajouter un contrôleur

- On utilise la commande suivante pour créer un contrôleur :

symfony console make:controller NomController

- ✓ L'exécution de la commande génère le contrôleur **NomController** avec une vue.
- ✓ L'action sur laquelle on mappe la route « /nom » peut retourner une vue, un texte, un json, ...

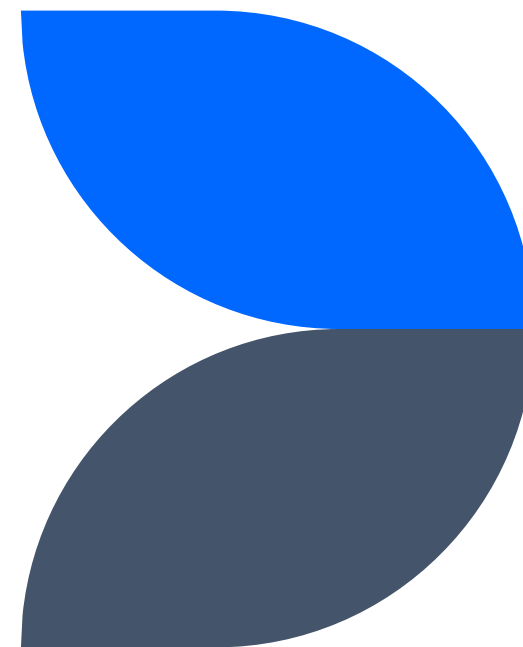
```
class NomController extends AbstractController
{
    #[Route('/nom', name: 'app_nom')]
    public function index(): Response
    {
        return $this->render('nom/index.html.twig', [
            'controller_name' => 'NomController',
        ]);
    }
}
```

Render() : Méthode de la classe **AbstractController** qui permet de rendre une vue

Création des routes avec des attributs

- ❖ Pour créer une route avec les attributs, on utilise **#[]**
- ❖ Une route est définie par des paramètres et des contraintes pour laquelle l'action se déclenche.
- ❖ Parmi les paramètres de la route :
 - La route en elle-même qui accepte des expressions régulières;
 - Le nom de la route,
 - la méthode d'accès HTTP doit être GET

**Template
Service TWIG**



Les Services

- Un service est une classe PHP globale qui remplit une fonction bien spécifique (envoyer des mails, effectuer une tâche, etc.)
 - Permet de séparer la logique métier dans l'application (on allège le contrôleur de ce qu'il n'est pas censé effectuer, il fera seulement appel aux services)
- Exemples de services : Twig, Templating, Kernel, Profiler, Session
- Pour voir les services par défaut : **symfony console debug:container**
- Pour voir tous les services : **symfony console debug:autowiring**

Le service Twig - Introduction

- Twig est un moteur de Template qui permet de séparer le code PHP (au niveau du contrôleur) du code HTML (au niveau des vues)
 - Simplifie l’affichage des données et le rend plus lisible
 - Système d’héritage très utile pour l’organisation des vues

Le service Twig – Fonctionnement(1/3)

- Template TWIG utilise **l'héritage**
- Un fichier père appelé « base.html.twig » sera à la racine du dossier « templates »

=> les éléments dans ce fichier seront communs à tous les autres templates TWIG en ajoutant la ligne
`{% extends "base.html.twig" %}`

- Cela fait gagner un temps fou de bien savoir exploiter ces « couches » proposées par TWIG

Le service Twig - Fonctionnement (2/3)

La Template de base est composée de plusieurs blocs :

- JavaScript
- Feuilles de styles CSS
- Corps contient le contenu de la page

```
<!DOCTYPE html>

<html lang="">
<head>
  <meta charset="UTF-8">
  <title>{% block title %}Welcome!{% endblock %}</title>
  {% block stylesheets %}{% endblock %}
</head>
<body>
  {% block body %}{% endblock %}
  {% block javascripts %}{% endblock %}
</body>
</html>
```

Le contenu de ces bornes sera remplacé par les **titres** spécifiés dans chaque page

Entre ces bornes on met les liens pour des **stylesheets**

Entre ces bornes c'est le contenu du **body** (dans la base on peut ajouter quelque chose avant ce block pour que ça apparaisse sur TOUTES les pages de notre site)

Entre ces bornes on met les liens pour le **javascript**

Le service Twig - Fonctionnement (3/3)

Lors de la création d'un contrôleur, on génère la vue avec laquelle on renvoie la réponse à l'utilisateur

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

On **hérite** du contenu de base.html.twig

```
{% block title %}Titre de la page{% endblock %}
```

On donne un **titre** à la page

```
{% block body %}  
  <h1>Voici la vue</h1>  
  <p>Ce contenu ne concerne que la page index.html.twig, qui hérite de base.html.twig</p>  
{% endblock %}
```

Contenu du **body** pour cette page

Pour éviter dans chaque Template de réécrire la structure HTML (head, body,...), Twig nous permet d'hériter la structure HTML directement via la Template de base en utilisant {% extends %}

Le service Twig - Syntaxe (1/3)

Afficher une variable

- `{{ }}` : permet d'afficher le contenu d'une variable ou d'une quelque chose

exemple : `{{2+2}}`, `{{ new date()}}`

Concaténation

- `~` : permet de faire la concaténation

ex : `{{ 'bon'~'jour'}}`

Documentation Twig Symfony : <https://twig.symfony.com/doc/2.x/>

Le service Twig - Syntaxe (2/3)

`{{ nom_variable | filtre1 | filtre2 | ... }}` : permet d'appliquer les filtres sur une variable

Exemples :

- ❖ `{{ name | upper | lower }}` : mettre en majuscule puis en minuscule le contenu de la variable **name**
- ❖ `{{ "now" | date }}` : formater la date actuelle avec le filtre **date**
 - ✓ `{{ "now" | date('h:i') }}` ⇔ `{{ "" | date('h:i') }}`
 - ✓ `{{ "now" | date('h:i A', "Europe/Paris") }}` : Date avec l'heure du jour avec zone Time et A : PM/AM
 - ✓ `{{ "yesterday" | date('d/m/y h:i:s') }}`
 - ✓ `{{ "tomorrow" | date('d/m/y h:i:s') }}`
 - ✓ `{{ "now" | date('h:i A') }}` : Année en cours
 - ✓ `{{ "now" | date('Y') }}`

```
$currentTime = (new DateTime('now'))->format('d/m/y h:m:s');//$currentTime = (new DateTime())->format('d/m/y h:m:s');  
$yesterday = (new DateTime('yesterday'))->format('d/m/y h:m:s');  
$tomorrow = (new DateTime('tomorrow'))->format('d/m/y h:m:s');
```

```
{{ "" | date('d/m/y h:i:s') }} <br/>  
{{ "now" | date('d/m/y h:i:s') }} <br/>  
{{ "yesterday" | date('d/m/y h:i:s') }} <br/>  
{{ "tomorrow" | date('d/m/y h:i:s') }} <br/>
```

Contrôleur ⇔ vue

C/C : twig peut gérer les dates => on peut enlever les variables dates déclarées dans le contrôleur

Le service Twig - Syntaxe (3/3)

❖ Afficher l'attribut d'un objet

```
{{ user.email }}
```

❖ Déclarer un bloc

```
{% block block_name %}
```

```
...
```

```
{% endblock %}
```

❖ Condition

```
{% if ... %} ...
```

```
{% elseif %} ...
```

```
{% else %} ...
```

```
{% endif %}
```

❖ Boucle

```
{% for user in listUsers %}
```

```
...
```

```
{% endfor %}
```

❖ Appeler une route (par exemple dans une barre de navigation) :

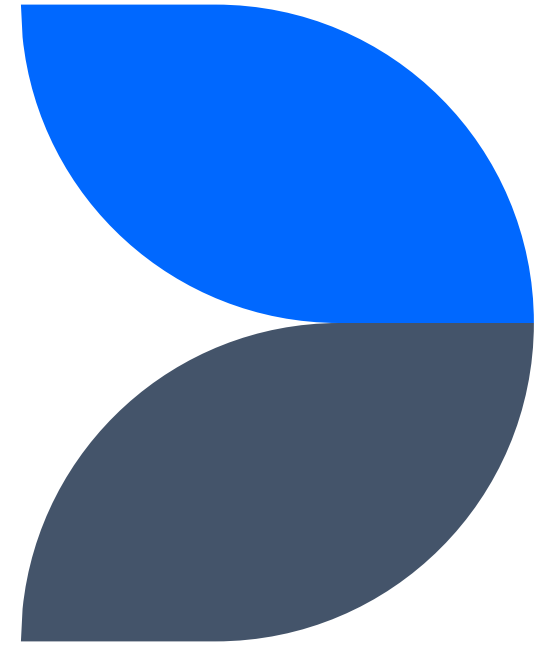
```
<a href="{{path('','parametre':'valeur')}}">lien</a>
```

❖ Lien vers un doc public :

```

```

**Communication entre un
contrôleur et une vue**



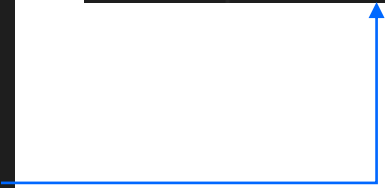
Contrôleur passe des variables à la vue(1/4)

Le contrôleur peut passer des variables à la vue. Ce dernier va se charger de les afficher.

Méthode 1 : Pour passer une variable à la vue, il faut la définir explicitement comme un paramètre dans la fonction **render()**

```
HomeController.php U X
c > Controller > HomeController.php > ...
1  <?php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6
7  class HomeController extends AbstractController{
8
9      //symfony appelle une méthode de la classe une action
10
11     public function index(){
12
13         $name = 'sana el atchia';
14         return $this->render('home.html.twig', ['xxx'=>$name]);
15     }
16 }
17
18 }
```

```
home.html.twig U X
templates > home.html.twig
1
2  {{xxx}}
3
```

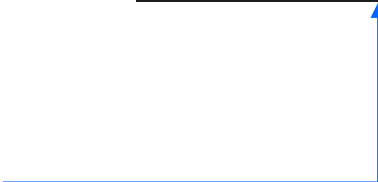


Contrôleur passe des variables à la vue (2/4)

Méthode 2 : Utilisation de la fonction **compact()** pour passer les variables dans les paramètres

```
HomeController.php U X
src > Controller > HomeController.php > ...
1  <?php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\HttpFoundation\Response;
6
7  class HomeController extends AbstractController{
8
9
10     public function index(){
11
12         $name = 'sana el atchia';
13
14         return $this->render('home.html.twig', compact(['name']));
15     }
16
17
18 }
19
```

```
home.html.twig U X
templates > home.html.twig
1
2  {{name}}
3
```



Contrôleur passe des variables à la vue (3/4)

```
<?php
namespace App\Controller;

use DateTime;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class HomeController extends AbstractController{

    //symfony appelle une méthode de la classe une action

    public function index(){

        $name = 'sana el atchia';
        // convertir la date en chaine de caractere
        $currentTime = (new DateTime())->format('d/m/y h:m:s');

        return $this->render('home.html.twig', compact(['name', 'currentTime']));

    }

}
```

```
home.html.twig U X
templates > home.html.twig
1
2 {{name}}
3 {{currentTime}}
```

Contrôleur passe un tableau de valeurs à la vue (4/4)

```
class NomController extends AbstractController
{
    #[Route('/nom', name: 'app_nom')]
    public function index(): Response
    {
        $users=["Adam","Robert","John","Susan"];
        return $this->render(view: 'nom/index.html.twig', parameters: compact('users',
        ));
    }
}
```

```
templates > nom > index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Utilisateurs{% endblock %}
4
5  {% block body %}
6  <div >
7      <ul>
8          {% for user in users %}
9              <li>{{user}}</li>
10             {% endfor %}
11         </ul>
12     </div>
13 {% endblock %}
```

Exercice

Créer cette page web en utilisant Symfony 😊

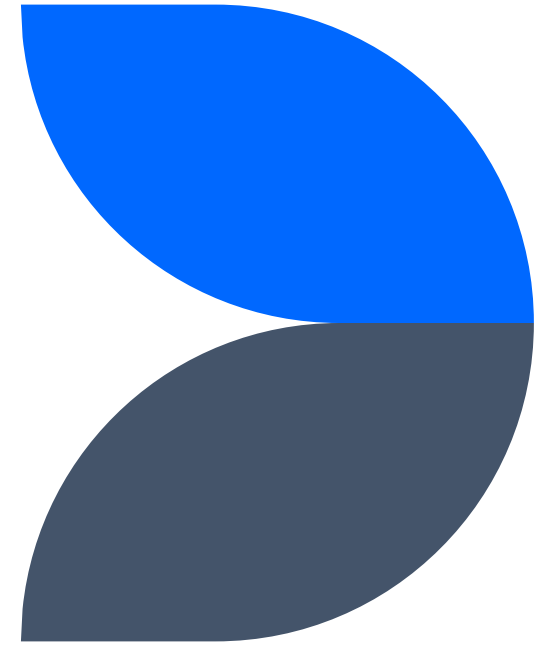


Bonjour tout le monde

Il est actuellement 05:38 PM à paris

Copyright 2023

Base de données
Modèle



Création de la base de données

1. Configurer la base de données

On configure la base de données (Mysql, Postgresql,...) dans le fichier **.env**

2. Créer la base de données

Pour créer la BD, on utilise la commande suivante :

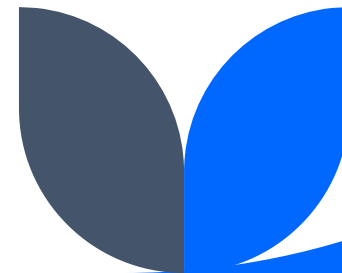
symfony console doctrine:database:create

ou

symfony console d:d:c

Object Relational Mapping (ORM)

- ❖ O.R.M est une technique de programmation faisant le lien entre le monde de la base de données et le monde de la programmation objet.
- ❖ O.R.M permet de créer une correspondance entre un **modèle objet** et un **modèle relationnel de base de données**.
- ❖ Un ORM sert à offrir une couche d'abstraction de connexion à toutes les BD relationnelles (comme PDO) mais aussi des facilités pour réaliser les requêtes courantes sans descendre au niveau des requêtes SQL et pour générer automatiquement des entités dans le langage utilisé avec les getters et setters correspondants.
- ❖ [Doctrine](#) est l'ORM par défaut du Framework Symfony




Création d'une entité

La commande suivante permet de créer une entité : **symfony console make:entity**

=>Elle génère deux fichiers : l'**entité** et un **Repository** qui permet de faire des **requêtes de sélections** SQL sur l'entité en question.

Exemple :

Créer l'entité Catégorie qui correspond à la table suivante :

Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
id 	int			Non	Aucun(e)		AUTO_INCREMENT
nom_cat	varchar(255)	utf8mb4_unicode_ci		Oui	NULL		

Création d'une entité

Créer l'entité Catégorie avec la commande suivante: **php bin/console make:entity Catégorie**

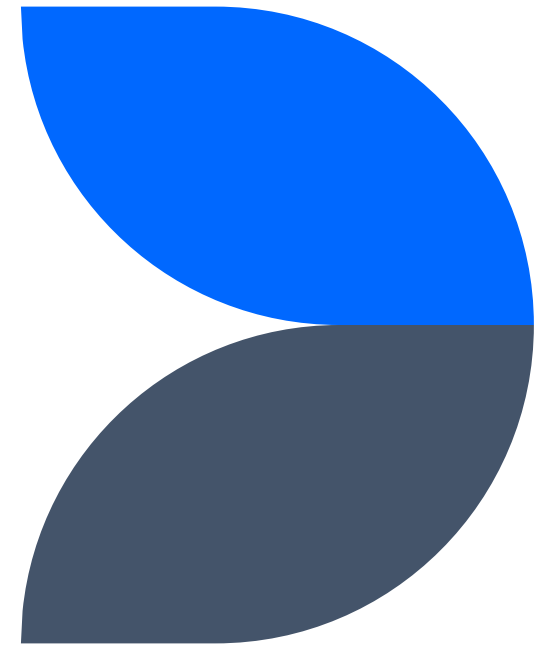
- New property name (press <return> to stop adding fields): **nomCat**
- Field type (enter ? to see all types) [string]: **string**
- Field length [255]: **255**
- Can this field be null in the database (nullable) (yes/no) [no]: **yes**

Remarque : L'entité Catégorie contient un id, un nomCat , les getters et les setters

Générer la table dans la base de données avec les commandes suivantes :

- ✓ **symfony console make:migration** : Génère le script SQL qui correspond à l'entité Catégorie
- ✓ **symfony console doctrine:migrations:migrate** ou **symfony console d:m:m** : permet d'exécuter les requêtes du fichier migration

Formulaire



Créer la classe Formulaire liée à une entité

En Symfony, les formulaires sont liés aux entités. Le but de cette partie est d'apprendre comment mettre en place des formulaires en Symfony.

- On va commencer par créer le formulaire d'une catégorie :

symfony console make:form CatégorieFormType Catégorie

- Le formulaire CatégorieFormType contient un champ input et un bouton :

```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

.....
$builder
    ->add('nomCat')
    ->add('sauvegarder', SubmitType::class);
```

Associer le formulaire à une route

- Affichez le formulaire dans une page web:
- Dans le dossier templates\categorie\, ajoutez une page newCategorie.html.twig
- Associez le formulaire à une route dans CategorieController.php :

```
use App\Entity\Categorie;
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Component\HttpFoundation\Request;
use App\Form\CategorieFormType;
.....
#[Route('/formCategorie', name: 'form_categorie')]
public function addCategorieForm(Request $request, ManagerRegistry $doctrine)
{
    $em = $doctrine->getManager();
    $cat = new Categorie();
    //créer un formulaire
    $form = $this->createForm(CategorieFormType::class, $cat);
    //gérer le traitement de la saisie
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){
        //dump($form->getData());
        $em->persist($cat);
        $em->flush();
        return $this->redirectToRoute('list_categorie');
    }

    return $this->render('categorie/newCategorie.html.twig', [
        'form' => $form->createView()
    ]);
}
```

Afficher la page Formulaire

Dans le fichier `newCategorie.html.twig`, ajoutez le code suivant :

```
{% extends 'base.html.twig' %}
{% block title %}Ajouter une catégorie{% endblock %}
{% block body %}

    <div >
        {# {{ form_start(form) }}
        {{ form_widget(form) }}

        {{ form_end(form) }} #}
        {{ form(form) }}

    </div>
{% endblock %}
```

Formulaire – Type d'un champ

Un formulaire est composé de champs, chacun étant construit à l'aide de un champ type (TextType, ChoiceType, ...). Le Framework Symfony offre une grande liste de types de champs pouvant être utilisés dans votre application.

Text Fields

- [TextType](#)
- [TextareaType](#)
- [EmailType](#)
- [IntegerType](#)
- [MoneyType](#)
- [NumberType](#)
- [PasswordType](#)
- [PercentType](#)
- [SearchType](#)
- [UrlType](#)
- [RangeType](#)
- [TelType](#)
- [ColorType](#)

Choice Fields

- [ChoiceType](#)
- [EnumType](#)
- [EntityType](#)
- [CountryType](#)
- [LanguageType](#)
- [LocaleType](#)
- [TimezoneType](#)
- [CurrencyType](#)

Other Fields

- [CheckboxType](#)
- [FileType](#)
- [RadioType](#)

Date and Time Fields

- [DateType](#)
- [DateIntervalType](#)
- [DateTimeType](#)
- [TimeType](#)
- [BirthdayType](#)
- [WeekType](#)

Hidden Fields

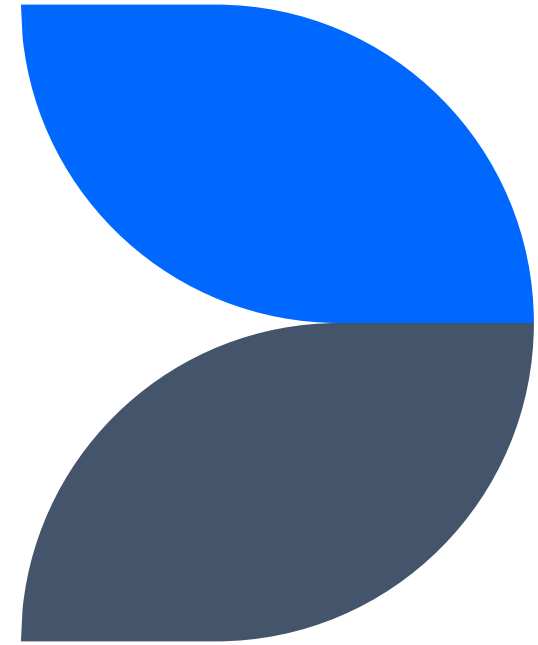
- [HiddenType](#)

Buttons

- [ButtonType](#)
- [ResetType](#)
- [SubmitType](#)

Lien : <https://symfony.com/doc/current/reference/forms/types.html>

Testing



Introduction

- ❖ Le test logiciel s'impose à toute l'équipe de développement afin de mesurer la qualité logiciel.
- ❖ Le test informatique favorise la détection de tous les bugs pour assurer la qualité de l'application.
- ❖ Le test logiciel garantit également l'acceptabilité du programme à la livraison et réduit la dette technique.
- ❖ On distingue les différents types de tests logiciel par niveaux et par nature.
 - Les tests peuvent être classés suivant leur nature :
 - ✓ **Le test fonctionnel** vise à garantir que chaque fonctionnalité de l'application fonctionne conformément aux exigences du logiciel. Il est souvent associé aux **tests de type boîte noire**
 - ✓ **Le test non fonctionnel** vise à vérifier les aspects non fonctionnels d'une application(robustesse, ergonomie, sécurité, ...). Il est souvent associé aux tests de **type boîte blanche** (par exemple les tests unitaires).



Tests logiciel par niveaux

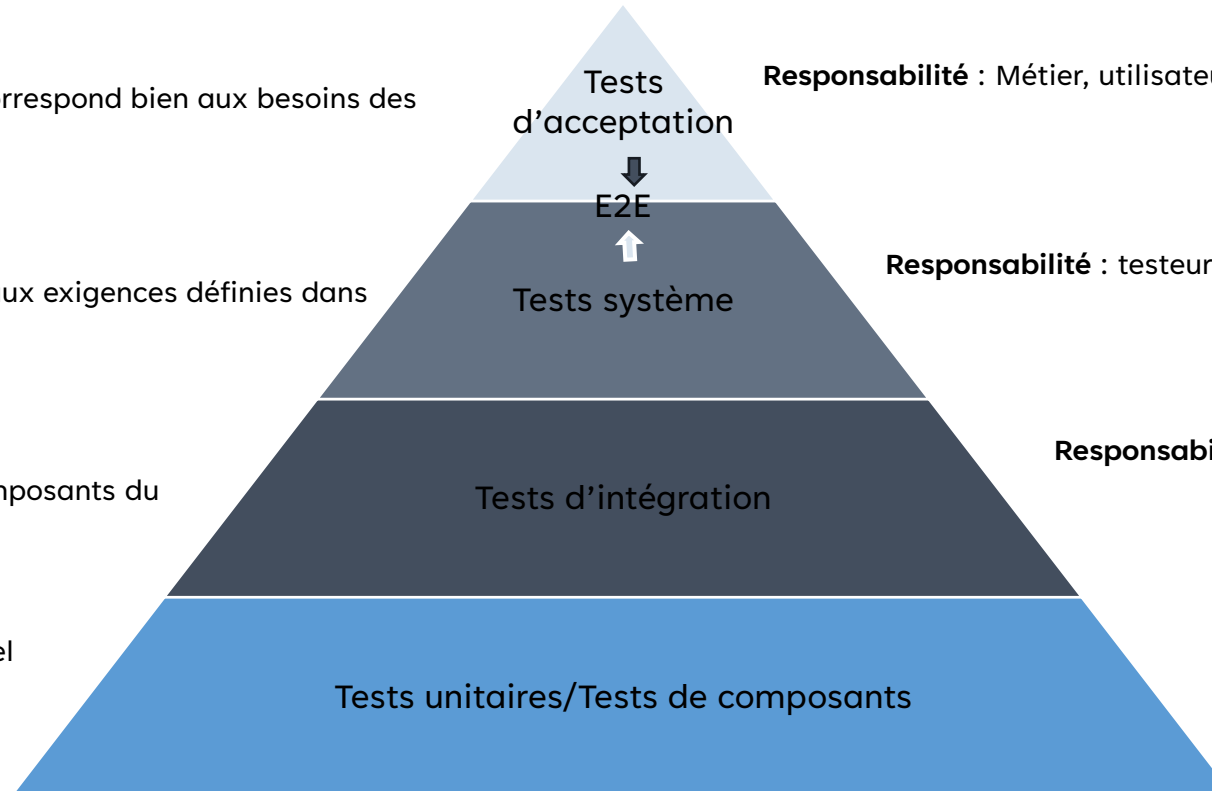
On parle de niveaux de test par rapport à la **pyramide des tests** suivante.

But : confirmer que le produit final correspond bien aux besoins des utilisateurs finaux.
100% manuelle

But : Vérifier que le système répond aux exigences définies dans les spécifications

But : Tester l'interaction entre les composants du logiciel

But : Tester les composants du logiciel séparément afin de s'assurer que chaque élément fonctionne comme spécifié



Responsabilité : Métier, utilisateur final

Responsabilité : testeur

Responsabilité : Développeur, Testeur technique

Responsabilité : Développeur

Tests appliqués en cours - Symfony

- Les tests unitaires
- Les tests fonctionnels

Mettre en place un outil pour implémenter des tests

- Pour faire les tests unitaires, nous allons utiliser l'outil PHPUnit.
- **PHPUnit** est la référence en matière de tests avec PHP. Il fournit toute une suite de classes et méthodes utiles pour faciliter l'écriture de tests.
- Installez PHPUnit avec Composer à partir du dossier où vous avez créé votre nouvelle application symfony.
composer require --dev phpunit/phpunit ^10
- Installez une dépendance supplémentaire pour lancer phpunit, qui s'appelle phpunit-bridge :
composer require --dev symfony/phpunit-bridge
- Vérifiez que PHPUnit est bien installé. Avec la commande suivante : **vendor/bin/phpunit** ou **php bin/phpunit**
- Lien vers la documentation du PHPUnit : <https://phpunit.de/documentation.html>
- NB : Le projet qu'on a créé ensemble contient déjà le PHP Unit, nous n'allons rien installer

Définition – Test unitaire

- Un test unitaire(UT,TU) est un bout de code qui va servir à tester une unité de code, à savoir une fonction.
- Le test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel / programme (appelée "unité" ou "module").
- Le principe d'un test unitaire est très simple. Il s'agit de :
 - ✓ Exécuter du code provenant de l'application à tester.
 - ✓ Vérifier que tout s'est bien déroulé comme prévu.
- **Les tests unitaires** sont réalisés à partir de la classe **TestCase** de PHPUnit.

Créer un test unitaire de base avec PHPUnit

- Dans le dossier tests, ajoutez deux dossiers : Unit & Functional
- Créez un test unitaire basique dans le dossier Unit avec la commande : `php bin/console make:test>>TestCase>>\App\Tests\Unit\BasicTest`
- Exécutez l'ensemble des tests de notre application avec la commande : `php bin/phpunit`
- Créez une base de données de test :
 1. Ajoutez un fichier avec le nom `.env.test.local` et mettez dedans : `DATABASE_URL="mysql://root:@127.0.0.1:3306/app_produit"`
 2. Ajoutez la BD de test(par défaut symfony ajoute `_test` au nom de la bd) avec la commande : `php bin/console d:d:c --env=test`
[Linux : `APP_ENV=test symfony console doctrine:database:create`]
 3. `symfony console d:migrations:migrate --env=test`
 4. Videz le cache dans environnement de test avec la commande : `php bin/console cache:clear --env=test`



Définition – Test fonctionnel

- Ce test vérifie l'intégration de différentes couches de l'application. Ils suivent un cycle de vie spécifique :
 1. Effectuer une requête auprès de l'application ;
 2. Parcourir le DOM (le code HTML de la page) ;
 3. Contrôler la réponse du serveur.

- Fonctionnement des tests fonctionnels dans Symfony :

Symfony utilise le composant **BrowserKit** construit autour des composants **DomCrawler** et **CssSelector**.

- **Le composant BrowserKit** : Simule le comportement d'un navigateur web, ce qui permet d'effectuer des requêtes, de cliquer sur des liens et de soumettre des formulaires de manière programmatique.
 - **Le composant DomCrawler** : Facilite la navigation dans le DOM pour les documents HTML et XML.
 - **Le composant CssSelector** : Convertit les sélecteurs CSS en expressions XPath.
- **Les tests fonctionnels** sont réalisés à partir de la classe **WebTestCase**, Cette classe étend la classe **KernelTestCase** qui elle-même étend de la classe **TestCase** de PHPUnit. Le rôle de ces classes est le suivant :
 - **La classe KernelTestCase** permet d'accéder au kernel de symfony.
 - **La classe WebTestCase** utilise le composant BrowserKit pour simuler un navigateur. Elle permet de créer une requête HTTP (HttpFoundation)



Créer un test fonctionnel –Tester une page

- Dans le dossier tests, ajoutez deux dossiers : Unit & Functional
- Créez un test unitaire basique dans le dossier Unit avec la commande : `php bin/console make:test>>WebTestCase>>\App\Tests\Functional\BasicTest`
- Exécutez l'ensemble des tests de notre application avec la commande : `php bin/phpunit`
- Modifiez le code du Tests\Functional\BasicTest :

```
public function testSomething(): void
{
    // $client simule un navigateur. Au lieu de faire des appels HTTP au serveur, il appelle directement l'application Symfony
    $client = static::createClient();
    $crawler = $client->request('GET', '/accueil');
    // Validation reponse 😊 + contenu

    $this->assertResponseIsSuccessful();
    $this->assertSelectorTextContains('h1', 'Bonjour tout le monde');
}
```


Tester l'entité Catégorie

- Créez un test de type KernelTestCase qui nous offre des services notamment validator qui va nous permettre de valider une entité :

`php bin/console make:test` >> KernelTestCase >> \App\Tests\Unit\CategorieTest

- Modifiez Tests\Unit\CategorieTest.php :

```
public function testEntiteCat(): void
{
    self::bootKernel();
    $container = static::getContainer();
    //utiliser des services> on va récupérer le composant validator puis on va compter les erreurs
    $cat = new Categorie();
    $cat->setNomCat("Décoration");
    //récupérer les erreurs
    $erros = $container->get('validator')->validate($cat);
    //demander que les errors soit 0
    $this->assertCount(0,$erros);
}
```

Tapez la commande : `php bin/phpunit`