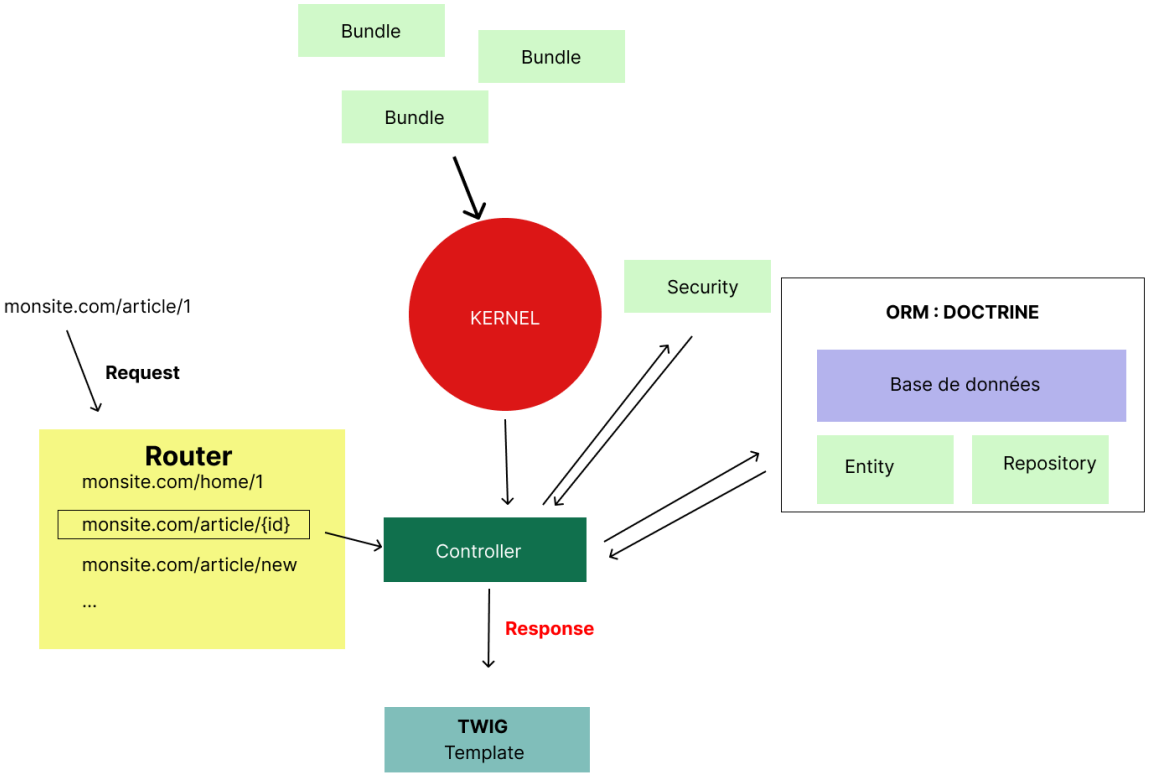
Les Controllers

> 👉 Les controllers servent à exécuter une action puis souvent rendre un template (ou envoyer une information, ou encore rediriger vers une page) en fonction de l'url courante. Un controller et par extension une méthode de controller sera appelée grâce à sa route (une url ou un template d'url) et retourne obligatoirement un objet de type Response

tous les controlleurs de symfony héritent de la classe AbstractController qui donne accès a tout un tas de méthodes bien utiles

Comment ca marche



Créer un Controller

pour créer un contrôleur et sont dossier de templates associes sans rien coder utiliser la commande suivante et répondre aux questions qui vous sont posées dans le terminal

⚠ Prenez l'habitude de la nomenclature des fichiers dans symfony, presque tout est Objet, et doit se nommer comme un objet : imaginons un contrôleur qui s'occupe de gérer les produits d'un magasin , il sera commun de l'appeler ProductController , il contiendra les méthodes a votre scénario comme par exemple une méthode list, new, edit, delete ,show ... pour un CRUD classique

```
symfony console make:controller
```

→ remarquez , symfony a automatiquement crée dans le dossier templates, un dossier spécial a partir du nom du controller , pour accueillir les templates utilisés par ce controller, on reviendra la dessus un peu plus tard

Les routes

Un controller peut avoir une route générale, plus communément on déclarera les routes grâce aux attributs php exemple :

```
#[Route('/list', name='app_list', methods=['GET'])]
public function list(): Response{
    //do something
}
```

Symfony permet d'utiliser des templates d'url pour y faire apparaitre des parametres variables comme par exemple :

```
#[Route('product/show/{id}', name='app_list', methods=['GET']
```

Dans ce cas symfony comprendra directement que le 3 morceau de la route est un id et de plus qu'il s'agit d'un id du model "product" !!! c'est fort n'est ce pas !

Les méthodes utiles de l'AbstractController

(à compléter ensemble)

- méthode json : répond du json on lui donnera en paramètre des données sous la forme d'un tableau associatif php (transformer un tableau en json⇒ json_encode())
- méthode render : rendre un template , 2 paramètres: le chemin du template, les paramètres qu'on envoie a notre template (un tableau associatif)
- méthode redirectToRoute : redirige vers un autre controller, prend en paramètre le nom de la route (name), le code HTTP
- getUser() : renvoie les informations de l'utilisateur connecté
- isGranted($role) : tester si un utilisateur est connecté (un utilisateur connecté a toujours un ROLE_USER
- addFlash($type,$message) prend en parametre un type (warning, success, danger...) et le message en question, ca donne acces dans nos templates a un tableau de messages destiné aux utilisateurs.

L'objet Request

L'objet `Request` est automatiquement injectable dans vos contrôleurs. Pour l'utiliser, il suffit de le déclarer en argument de votre méthode :

```
<?php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractControl
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

class MyController extends AbstractController
{
        #[Route(.....)]
    public function myAction(Request $request): Response
    {
        // Utilisation de $request ici
```

```
    }
}
```

⚠ lorsque vous importez les classes avec le use faite bien attention au namespace des classes importée , ici pour les objets Request et Response selectionnez bien le namespace

Symfony\Component\HttpFoundation\

Récupération des informations de la requête

Méthode HTTP

- `$request->getMethod()` : Retourne la méthode HTTP utilisée (GET, POST, PUT, DELETE, etc.).

URL

- `$request->getRequestUri()` : Retourne l'URL complète de la requête.
- `$request->getPathInfo()` : Retourne la partie de l'URL après le nom de domaine.

Paramètres

- `$request->query->get('param')` : Récupère un paramètre de la chaîne de requête (GET).
- `$request->request->get('param')` : Récupère un paramètre envoyé dans le corps de la requête (POST, PUT, etc.)..

En-têtes HTTP

- `$request->headers->get('header_name')` : Récupère la valeur d'un en-tête HTTP.

Contenu de la requête

- `$request->getContent()` : Retourne le contenu brut de la requête (utile pour les requêtes POST avec un corps).

Test de la méthode HTTP

PHP

```
if ($request->isMethod('POST')) {
    // Traitement spécifique aux requêtes POST
} else {
    // Traitement spécifique aux autres méthodes
}
```

Autres méthodes utiles

- `$request->isXmlHttpRequest()` : Vérifie si la requête est une requête AJAX (XMLHttpRequest).
- `$request->getSession()` : Accède à la session de l'utilisateur.

Automatiquement injecté ? ⇒ l'injection de dépendances

Petite habitude a prendre avec le framework lorsque l'on veut utiliser une classe on peut directement la déclarer en paramètre d'une méthode et importer la classe (use)

Automatiquement une instance de cette classe sera crée si elle le peut et vous aurez accès dans le corps de votre fonction a toutes les méthodes de cette classe.

Exercice :

Exercice 1 :

créer un controller qui permet l'affichage des données d'un utilisateur dans un template twig

```
$utilisateur = [
'id' => 123,
'nom' => 'Jean Dupont',
'email' => 'jean.dupont@exemple.com',
'date_naissance' => '1980-01-01',
```

```
'adresse' => [
    'rue' => 'Rue de la Paix',
    'code_postal' => '75002',
    'ville' => 'Paris'
],
'est_actif' => true
];
```

Exercice 2:

écrire une méthode de ce Controller qui envoie une donnée JSON simple lorsque la route de celle ci est appelée en ajax depuis un script javascript en méthode GET

```
$couleurs = [
'rouge' => '#FF0000',
'vert' => '#00FF00',
'bleu' => '#0000FF',
'jaune' => '#FFFF00',
'noir' => '#000000'
];
```