



# Symfony



High Performance PHP Framework for  
web development



# Back to Basics

---

# La POO

## Programmation Orientée Objet

- Glossaire
- Encapsulation
- Héritage
- Polymorphisme
- Interfaces



# Glossaire

---

- *Objet* : Structure de données valuées qui répond à un ensemble de messages. Un objet est représenté par une *Classe* ou une *Interface*
- *Attribut* : Donnée ou Objet qui représente l'objet
- *Méthode* : Action de l'objet
- *Constructeur* : Méthode appelé lorsque l'objet est créé
- *Type* : Texte qui représente l'objet
- *Instance* : Représentation concrète d'un objet

# Encapsulation

---

Permet de définir la visibilité des attributs et des méthodes de l'objets

- *public* : Tout le monde peut voir l'attribut ou la méthode
- *private* : Seul l'objet connaît l'attribut ou la méthode
- *protected* : Seul l'objet et son/ses héritier/s connaissent l'attribut ou la méthode
- *package* : Tous les objets du "package" connaissent l'attribut ou la méthode (très peu utilisé)

# Encapsulation

---

## public

Pour un objet *Homme* :

- *la couleur des cheveux* est un attribut public
- *manger()* est une méthode public

# Encapsulation

---

## private

Pour un objet *Femme* :

- *date de naissance* est un attribut privé
- *donnerNaissance()* est une méthode privée

# Encapsulation

---

## protected

Pour un objet *Homme* :

- *nombre de dents* est un attribut protected hérité de l'objet *Humain*
- *marcherDebout()* est une méthode protected héritée de l'objet *Humain*



# Héritage - Définition

---

Un objet dit "père" peut transmettre ses caractéristiques *public* et *protected* à son(ses) objet(s) dit "fils".

On dit que l'objet dit "fils" *hérite* de l'objet dit "père".

Dans la littérature, on dira aussi de façon synonymique que :

- le "fils" *étend* (*extends*) le "père"
- le "fils" est une *spécialisation* du "père"
- le "fils" *dérive* du "père"

Cela est aussi appelé *Polymorphisme par héritage*.

# Héritage - Exemple

---

Exemple :

Un objet **Femme** héritera des caractéristiques d'un objet **Humain**.

Un objet **Homme** héritera aussi des caractéristiques d'un objet **Humain**.

Mais chaque objet **Femme** et **Homme** a des attributs et méthode spécifiques.

# Polymorphisme

---

Le *polymorphisme* est la possibilité d'un attribut, d'une méthode ou d'un objet de prendre plusieurs formes.

Il existe plusieurs types de polymorphisme. Voici les 2 essentiels :

- *Polymorphisme par héritage* permet dans une classe dite "fille" de redéfinir une méthode héritée
- *Polymorphisme paramétrique* permet d'avoir le même nom de méthode mais avec des paramètres différents, que ce soit en nombre ou en type. *Cela n'est pas possible en PHP*

# Polymorphisme

---

## Polymorphisme par héritage

Exemple :

Si la classe **Homme** hérite de la méthode **parler()**, alors la classe pourra redéfinir la méthode **parler()** pour y introduire la notion de la mue de la voix.

# Classe, Classe Abstraite et Interface

---

Un objet dont on veut avoir une ou plusieurs instance sera défini en tant que *Classe*.

En PHP, cela donnera :

```
class Homme {  
}
```

# Classe, Classe Abstraite et Interface

---

Un objet dont on ne veut pas avoir d'instance sera défini en tant que *Classe Abstraite*.

En PHP, cela donnera :

```
abstract class AbstractHumain {  
}
```

```
class Homme extends AbstractHumain {  
}
```

# Classe, Classe Abstraite et Interface

---

Une Interface est un ensemble de signatures de méthodes publiques d'un objet.

En PHP, cela donnera :

```
interface IMammal {  
    public function respirer();  
}
```

```
abstract class AbstractHumain implements IMammal {  
  
}
```

# Static

---

*static* est un type particulier qui peut être attribué aux *attributs* et *méthodes* d'un *objet*.

Un attribut ou une méthode *static* sera accessible même si l'objet n'est pas instancié.



# Static

---

```
class Homme {  
    public static $couleurCheveux = "noir";  
    public static function marcher() { /* ... */ }  
}
```

*// Le code suivant fonctionnera dans n'importe quelle méthode*

```
Homme::$couleurCheveux; // pas d'erreur car static  
Homme::marcher(); // pas d'erreur car static
```

# UML

Unified Modeling Language

- Diagramme de classe et d'objets



# Diagramme de classe et d'objets

---

Un diagramme de classe représente les classes avec les attributs et les méthodes, ainsi que la généralisation et les associations entre les classes.

# Diagramme de classe et d'objets

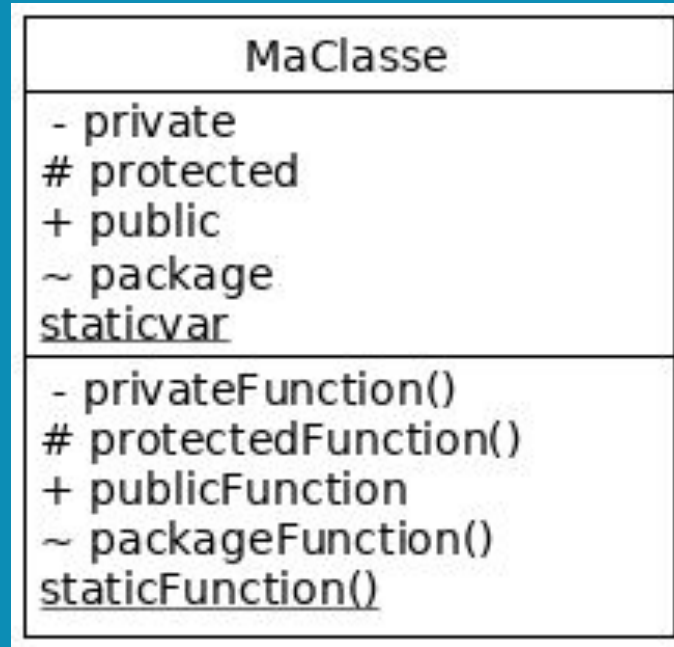
---

La représentation des *attributs* et *méthodes* est accompagné de leur visibilité.

- *private* est représenté par un -
- *protected* est représenté par un #
- *public* est représenté par un +
- *package* est représenté par un ~
- le type *static* est représenté par le souligné.

# Diagramme de classe et d'objets

---



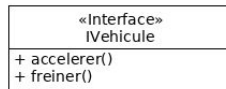
# Diagramme de classe et d'objets

---

Le diagramme de classe permet de représenter une interface.  
Pour cela, on dessine un rectangle avec un trait plein horizontal à l'intérieur.  
Dans la partie haute se trouve le nom de l'interface surplombé par «*Interface*».  
Dans la partie basse se trouve les signatures des méthodes.

# Diagramme de classe et d'objets

---



# Diagramme de classe et d'objets

---

Le diagramme de classe permet de représenter une classe abstraite. Pour cela, on dessine un rectangle avec deux traits pleins horizontaux à l'intérieur.

Dans la partie haute se trouve le nom de la classe abstraite surplombé par «*Abstract*».

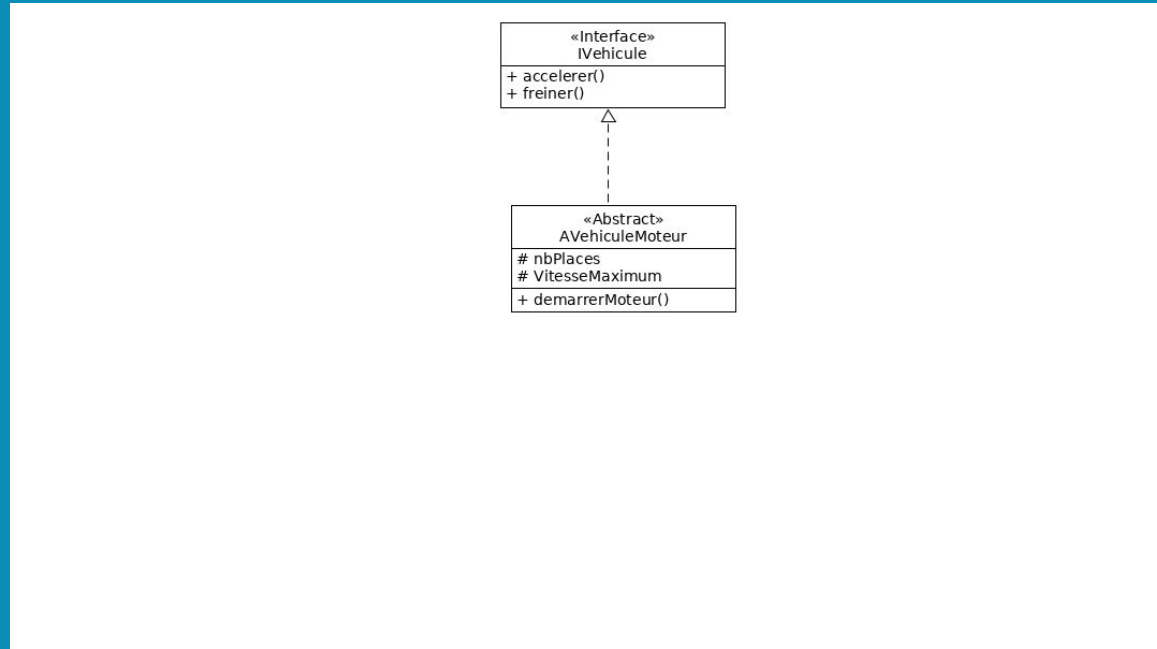
Dans la partie du milieu se trouve les attributs de la classe. Dans la partie basse se trouve les signatures des méthodes.

La notion d'implémentation d'une interface sera représentée par une *flèche vide fermée avec un trait en pointillée*.

Au bout de la flèche se trouve l'interface.



# Diagramme de classe et d'objets



# Diagramme de classe et d'objets

---

Le diagramme de classe permet de représenter une classe.

Pour cela, on dessine un rectangle avec deux traits pleins horizontaux à l'intérieur.

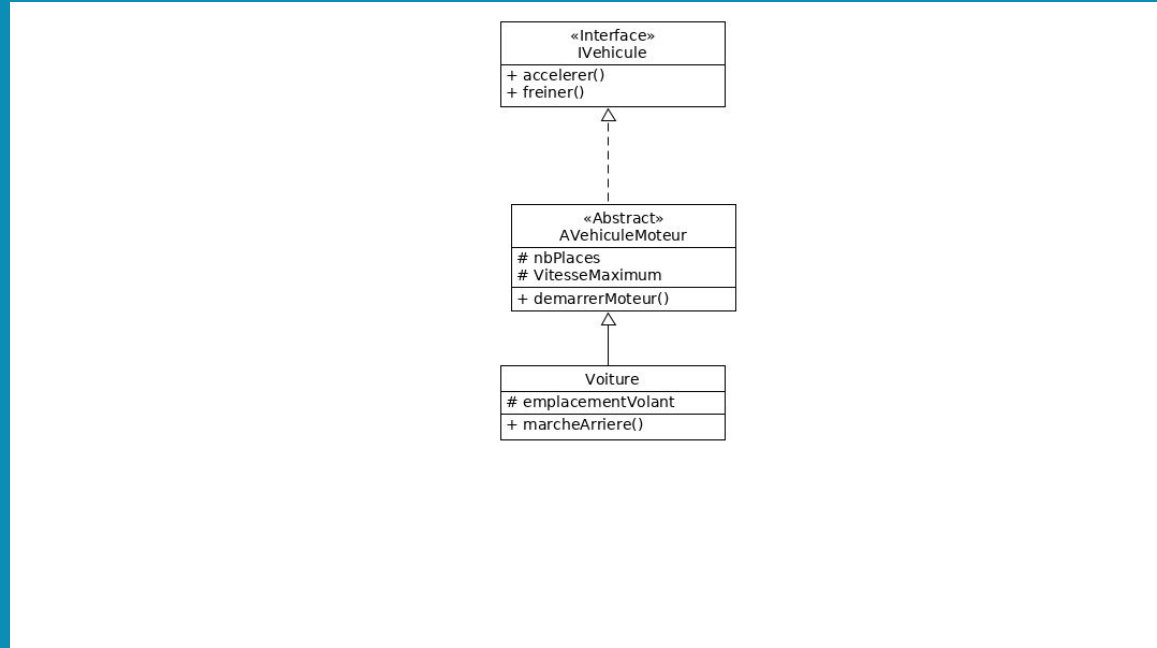
Dans la partie haute se trouve le nom de la classe.

Dans la partie du milieu se trouve les attributs de la classe. Dans la partie basse se trouve les signatures des méthodes.

La notion d'héritage d'une classe sera représentée par une *flèche vide fermée avec un trait plein*.

Au bout de la flèche se trouve la classe dont on hérite.

# Diagramme de classe et d'objets



# Diagramme de classe et d'objets

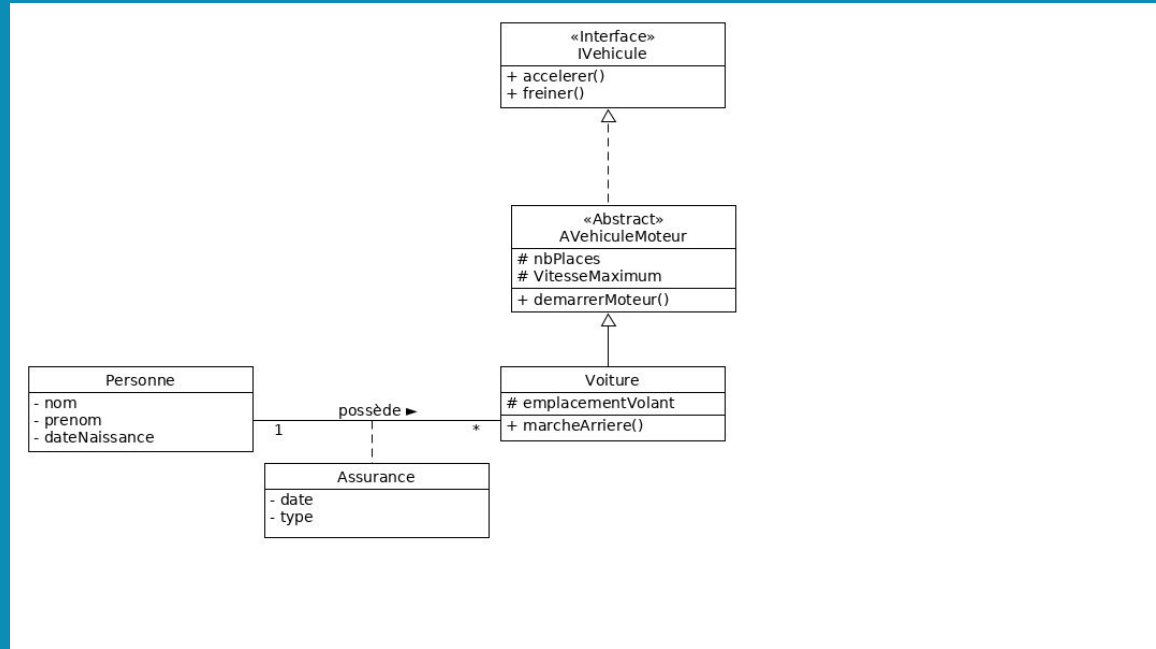
---

Une *association* de deux classes peut se faire en les reliant par un *trait plein* avec de chaque côté la cardinalité.

La cardinalité indique combien d'instance d'une classe peuvent être associées à une seule instance de l'autre classe.

Cette association peut être agrémentée d'un texte explicatif ainsi que d'une *classe-association*.

# Diagramme de classe et d'objets



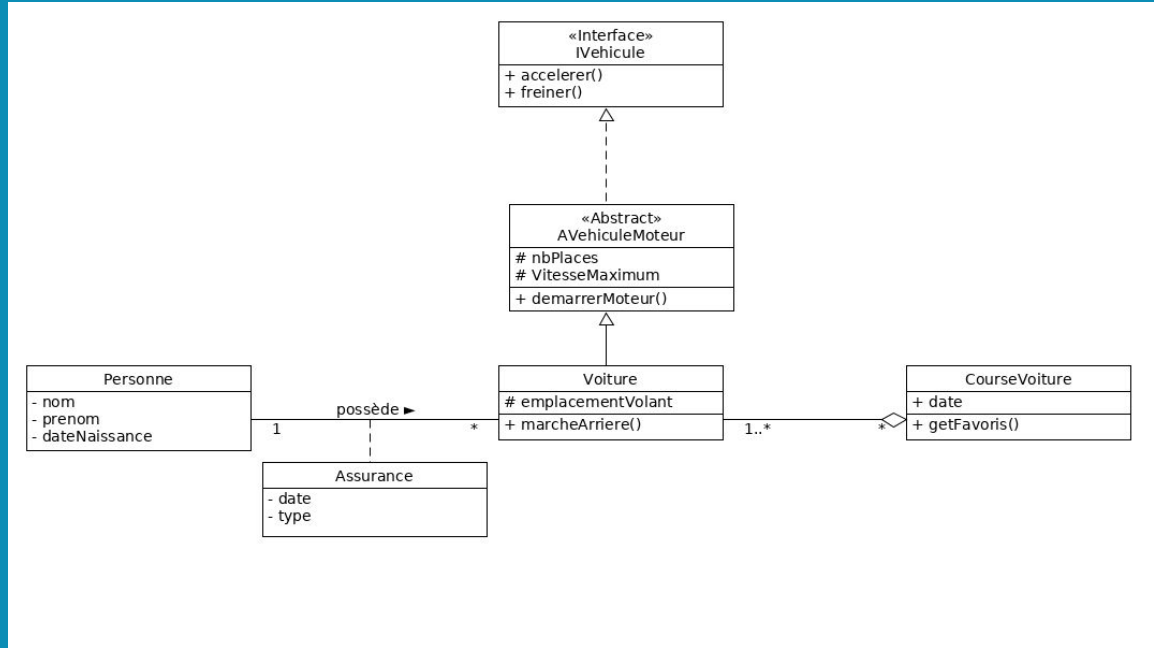
# Diagramme de classe et d'objets

---

Une *agrégation* correspond au fait qu'un objet (appelé ici *composé*) peut être composé de plusieurs objets, mais la destruction du premier **n'entraîne pas** la destruction de ces derniers.

Une agrégation est représentée par *un trait plein terminé par un losange vide du coté du composé*.

# Diagramme de classe et d'objets



# Diagramme de classe et d'objets

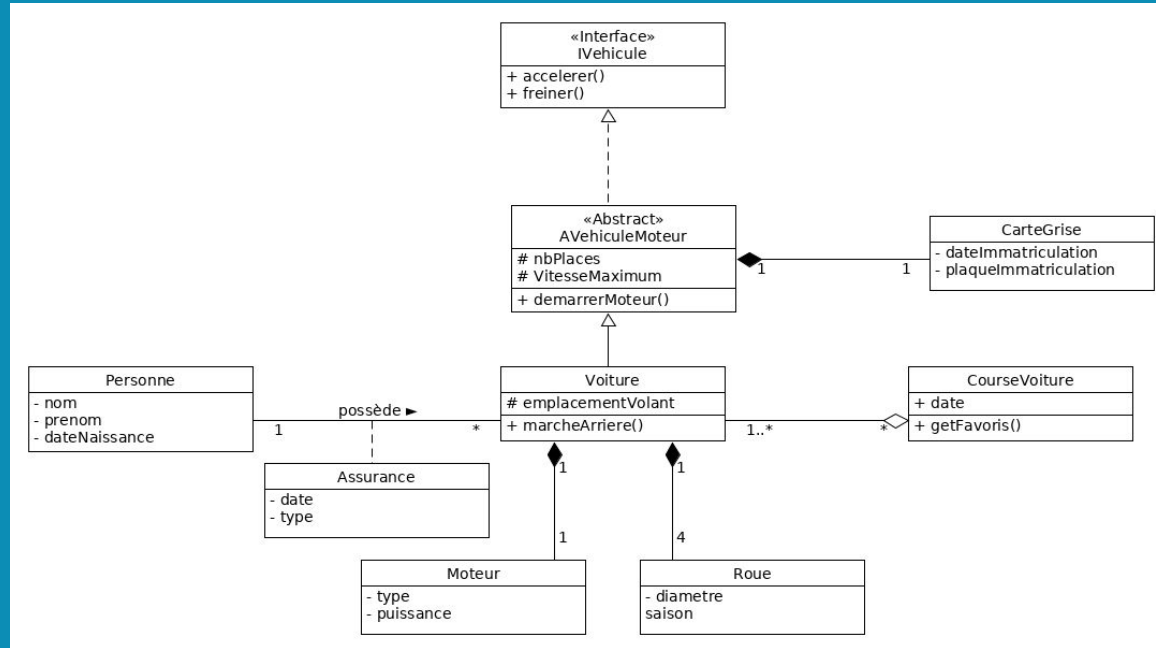
---

Une *composition* correspond au fait qu'un objet (appelé ici *composé*) peut être composé de plusieurs objets, mais la destruction du premier **entraîne** la destruction de ces derniers.

Une composition est représentée par *un trait plein terminé par un losange plein du côté du composé*.



# Diagramme de classe et d'objets



# Diagramme de classe et d'objets

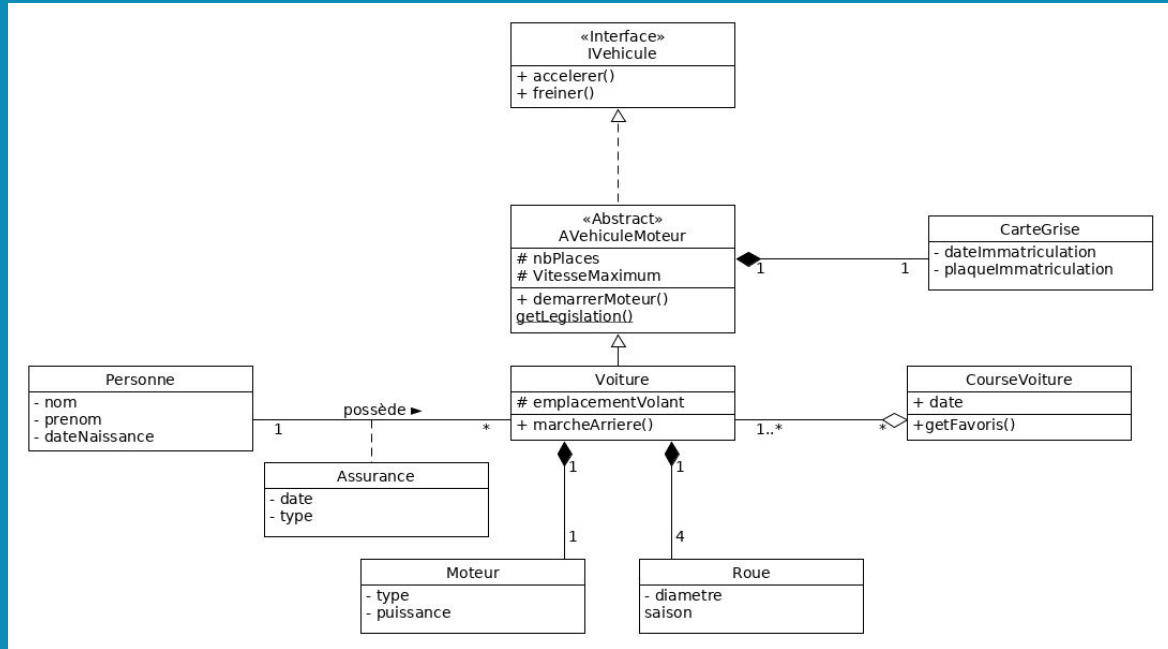
---

Pour terminer, nous allons indiquer la présence d'une méthode *static*.

Pour rappel, une méthode *static* peut être appelée même si l'objet n'est pas instancié.

Il en est de même pour une variable *static*.

# Diagramme de classe et d'objets

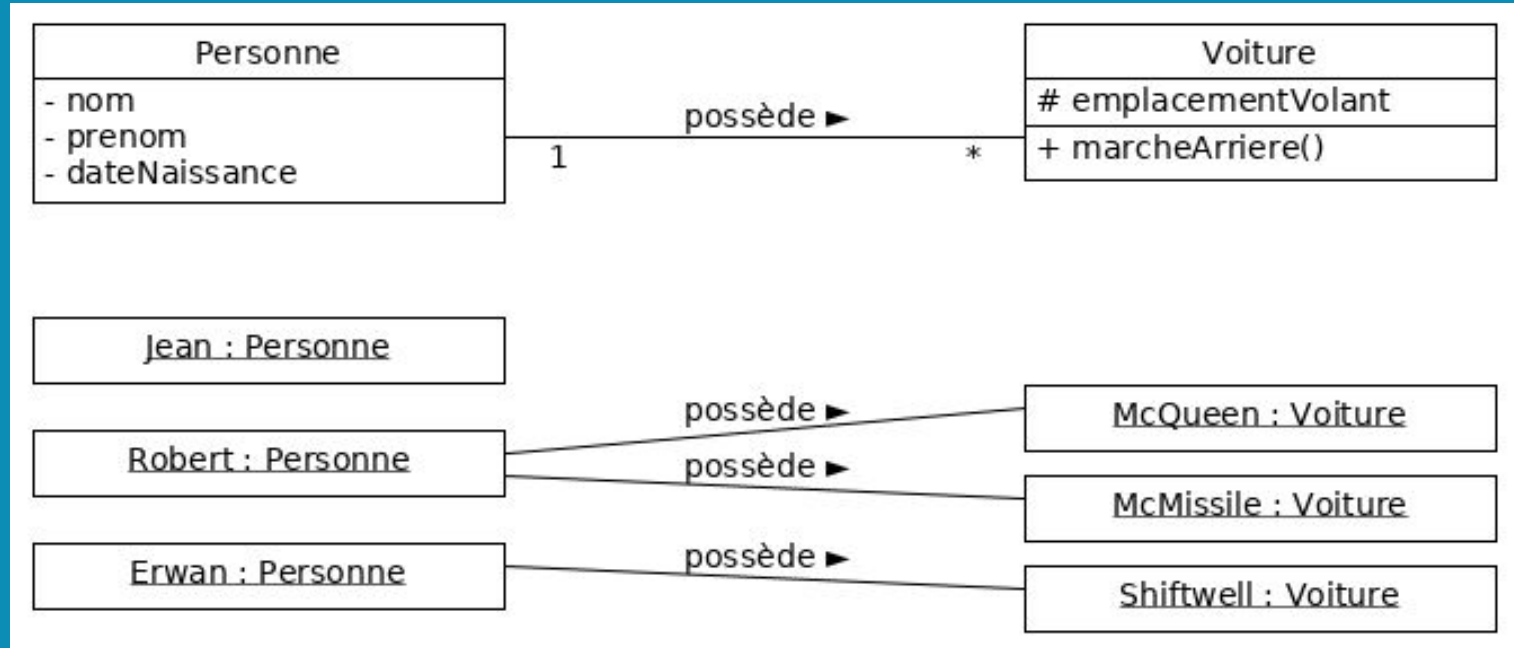


# Diagramme de classe et d'objets

---

Le diagramme objet permet de représenter concrètement l'impact des cardinalités.

# Diagramme de classe et d'objets



# TD *Diagramme de classe*

---

# Format de Données

Principaux formats utilisés dans les APIs

- XML
  - JSON
-

# XML

---

eXtensible Markup Language

Le XML fut créé en 1998.

Il est basé sur le SGML (qui a inspiré le HTML).

C'est un métalangage normalisé par le W3C.

La version 1.1 date de 2006.



# XML

---

*« Son but est de permettre au SGML générique d'être transmis, reçu et traité sur le Web de la même manière que l'est HTML aujourd'hui. »*

XHTML a été créé dans le but de remplacer le HTML... mais il a échoué.

XML est populaire parce qu'il est simple à écrire et à lire.  
Par contre, il est très verbeux.

# XML

---

```
<?xml version="1.1" encoding="UTF-8"?>
<article xmlns="http://docbook.org/ns/docbook">
  <title>Extensible Markup Language</title>
  <para>
    <acronym>XML</acronym> (Extensible Markup Language,
« langage de
balisage extensible »)...
  </para>
</article>
```

# JSON

---

JavaScript Object Notation

JSON permet de définir un objet en javascript.

Il a été créé en 2002 et 2 organismes concurrents (IETF et ECMA) l'ont normalisé.

# JSON

---

Un document JSON comprend deux types d'éléments structurels :

- des ensembles de paires « nom » (alias « clé ») / « valeur »
- des listes ordonnées de valeurs

Ces mêmes éléments représentent trois types de données :

- des objets
- des tableaux
- des valeurs génériques de type tableau, objet, booléen, nombre, chaîne de caractères ou null.

# JSON

---

Il est facile de passer d'un objet à une représentation JSON, et vice versa.

JSON est peu verbeux.

JSON est compris nativement par tous les navigateurs.

Par contre, JSON ne permet pas autant de chose que le XML (ex : balise en plein milieu d'un texte).

**JSON est le format le plus utilisé dans les APIs.**

# JSON

---

```
{
  "menu": {
    "id": "file",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```



**Questions?**

# Symfony

---



# Symfony

framework MVC Open Source.

- Historique
- Avantages
- Roadmap
- Alternatives



# Symfony ?

---

*Symfony is a set of PHP Components, a Web Application framework, a Philosophy, and a Community – all working together in harmony.*

<https://symfony.com>

<https://github.com/symfony/symfony>

# Historique Symfony

---

## Principales dates de symfony

1. Rendu open-source en 2005
2. Symfony 1 : 2007
3. Symfony 2 : 2011, Réécriture complète de Symfony
4. Symfony 3 : novembre 2015 – novembre 2017
5. Symfony 3.4 : novembre 2017 Version LTS qui a marqué le début d'une stabilisation de la Roadmap, introduction de Flex
6. Symfony 4 : novembre 2017, (R)Évolution dans la construction d'un projet symfony
7. Symfony 4.4/5 : novembre 2019, Version LTS

# Avantages Framework

---

- Bonnes pratiques
- accélère le développement de fonctionnalité récurrente
- mutualise la maintenance du socle de l'application
- a probablement déjà fait face à des problématiques complexes

# Avantages Open Source

---

- Tout le monde peut analyser/corriger/optimiser le code du framework
- Vente de service vs vente de produit
- Facile de s'auto-former et de former
- Même si SensioLabs ferme, Symfony continuera d'exister

# Avantages Symfony - part I

---

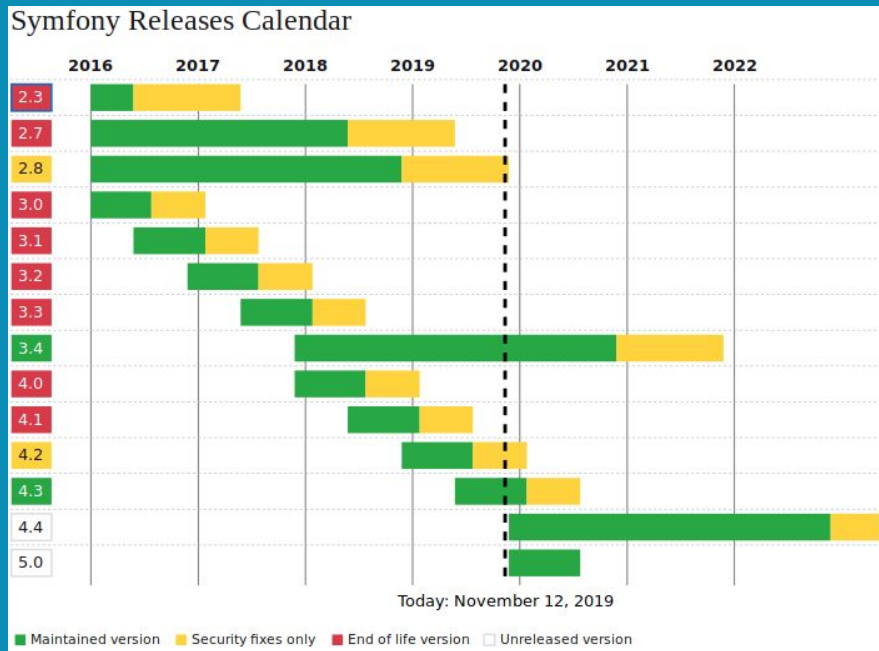
1. Symfony est très impliqué dans la communauté PHP (PHP-FIG)
2. Symfony regroupe une grande communauté en France
  - a. Meetup AfSY et très présent dans les Meetup AFUP
  - b. Slack Symfony <https://symfony.com/slack>
  - c. Stack Overflow Symfony : <https://stackoverflow.com/questions/tagged/symfony>
3. Symfony innove et suit les dernières innovations
  - a. Symfony Cloud
  - b. Malgré Twig, se concentre sur la partie Back et laisse le front aux frameworks JS

# Avantages Symfony - part II

---

1. Symfony est pro
  - a. SensioLabs (filiale de Smile) développe Symfony
  - b. Roadmap clair et suivie
  - c. LTS
2. Symfony (ou un autre framework php) est une compétence recherchée en France

# Roadmap





# Alternatives à Symfony

---

Framework :

1. Laravel : <https://github.com/laravel/laravel>
2. Zend Framework : <https://github.com/zendframework/zendframework>
3. CakePHP, ...

CMS, CRM, générateur de sites static, ...

Symfony est fait pour le sur-mesure !



# Fondations

standing on the shoulders of  
giants

- PHP-FIG
- Composer
- Packagist
- Flex
- Injection de dépendances



# PHP-FIG

PHP Framework Interop Group



# PHP-FIG

---

- PHP-FIG
  - PSR (PHP Standard Recommendation)
- Définit un ensemble de bonnes pratiques dans toutes la communauté PHP

-> Règles pour écrire le code

-> Règles pour namespaces

-> Règles pour autoload de classe

-> ...

PHP-cs-fixer Permet de formater automatiquement le code.

# Composer

A Dependency Manager for PHP



# Composer

---

<https://getcomposer.org>

- remplace PEAR
- contient un `composer.json` pour connaître les dépendances du projet
- contient un `composer.lock` pour connaître les dépendances installées par le projet (fixe le commit de chaque dépendance)
- **génère l'autoload qui sera chargé par le framework**
- on peut créer un projet avec `composer create-project symfony/skeleton`
- on peut exécuter des scripts à l'installation et à la mise à jour
- etc...

# composer.json

Tous les détails dans la doc :

<https://getcomposer.org/doc/04-schema.md>

Cheat Sheet de JoliCode :

<https://composer.json.jolicode.com/>





# composer.json

```
{
  "type": "project",
  "license": "proprietary",
  "require": {
    "php": "^7.1.3",
    "ext-type": "*",
    "ext-iconv": "*",
    "symfony/console": "4.3.*",
    "symfony/dotenv": "4.3.*",
    "symfony/finder": "4.3.1",
    "symfony/framework-bundle": "4.3.*",
    "symfony/yaml": "4.3.*"
  },
  "require-dev": {
  },
  "config": {
    "preferred-install": {
      "*": "dist"
    },
    "sort-packages": true
  },
  "autoload": {
    "psr-4": {
      "App\\": "src/"
    }
  },
  "autoload-dev": {
    "psr-4": {
      "App\\Tests\\": "tests/"
    }
  },
  "replace": {
    "paragonie/random_compat": "2.*",
    "symfony/polyfill-ctype": "*",
    "symfony/polyfill-iconv": "*",
    "symfony/polyfill-php71": "*",
    "symfony/polyfill-php70": "*",
    "symfony/polyfill-php56": "*"
  },
  "scripts": {
    "auto-scripts": {
      "cache:clear": "symfony-cmd",
      "assets:install %PUBLIC_DIR%": "symfony-cmd"
    },
    "post-install-cmd": [
      "@auto-scripts"
    ],
    "post-update-cmd": [
      "@auto-scripts"
    ]
  },
  "conflict": {
    "symfony/symfony": "*"
  },
  "extra": {
    "symfony": {
      "allow-contrib": false,
      "require": "4.3.*"
    }
  }
}
```

# Packagist

The PHP Package Repository



# Packagist

---

<https://packagist.org>

- Recense toutes les librairies PHP
  - Une librairie est un ensemble fonctionnel
  - Une librairie peut avoir ses propres dépendances
- Recense tous les bundles Symfony
  - Un bundle Symfony est une librairie destinée à être utilisée dans Symfony
  - Certains bundle sont des bridges avec une librairie
  - Un bundle peut avoir ses propres dépendances
- Composer va chercher dans Packagist les dépendances à installer

# Packagist

---

Exemple de librairie :

- <https://packagist.org/packages/google/apiclient> Librairie pour appeler les api google

Exemple de bundle :

- <https://packagist.org/packages/eightpoints/guzzle-bundle> Bundle Symfony pour configurer simplement l'utilisation de la librairie guzzle

# Flex

Symfony Recipes



Symfony Recipes Server

# Flex

---

Sans Flex, la configuration d'un bundle pose plusieurs problèmes :

- On ne peut pas proposer une configuration par défaut sans la forcer au développeur
- Lorsqu'on retire un bundle, sa configuration reste car elle est mélangée avec les confs des autres bundles

# Flex

---

LA nouveauté de symfony4.

Flex permet de créer des "recettes" associées à l'installation d'un bundle.  
Grâce à ces recettes :

- la configuration de chaque bundle est bien identifié
- Chaque bundle peut proposer une configuration par défaut
- À la désinstallation du bundle, la configuration est retirée

# Flex

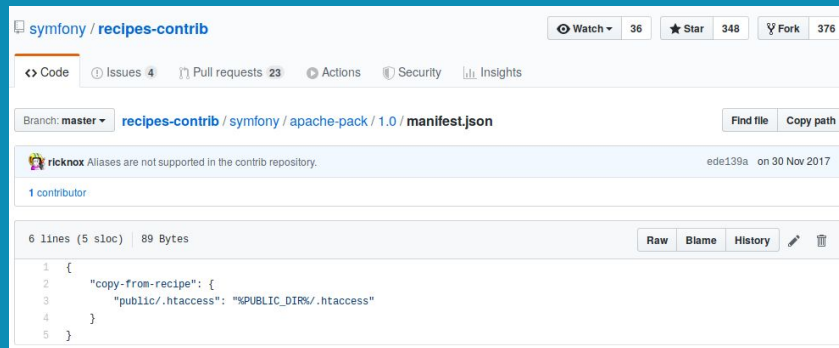
Flex étend Composer, et dit à composer d'aller chercher dans le repository github de Symfony recipes *avant* d'aller chercher dans Packagist.

Liste des recettes : <https://flex.symfony.com>

Repositories github des recettes :

<https://github.com/symfony/recipes> et

<https://github.com/symfony/recipes-contrib>



The screenshot shows the GitHub interface for the repository `symfony / recipes-contrib`. At the top, there are buttons for 'Watch' (36), 'Star' (348), and 'Fork' (376). Below these are tabs for 'Code', 'Issues' (4), 'Pull requests' (23), 'Actions', 'Security', and 'Insights'. The 'Code' tab is selected, showing the file path `Branch: master recipes-contrib / symfony / apache-pack / 1.0 / manifest.json`. A message from `ricknox` states: 'Aliases are not supported in the contrib repository.' with a commit hash `ede139a` on `30 Nov 2017`. Below this, it says '1 contributor'. The file content is displayed as 6 lines (5 sloc) and 89 Bytes. The code is a JSON object:

```
1 {  
2   "copy-from-recipe": {  
3     "public/.htaccess": "%PUBLIC_DIR%/.htaccess"  
4   }  
5 }
```



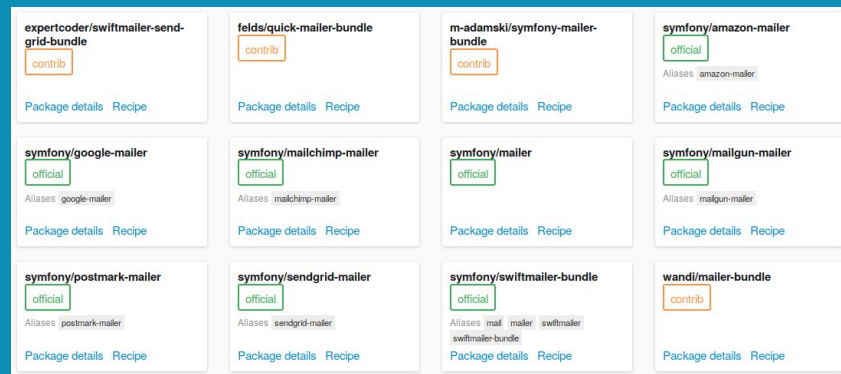
# Flex

## Critique :

Toutes les recettes ne sont pas au même niveau. Il y a les *officielles* et les *contrib*.

Concrètement, les officielles peuvent avoir à un alias, et pas les contrib.

Le niveau de qualité des officiels est plus élevé.



# TD *Création d'un projet avec Symfony*

---

# Injection de dépendance

---

*L'injection de dépendances (dependency injection en anglais) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle.*

*Il consiste à créer dynamiquement (injecter) les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.*

source : [wikipedia](https://fr.wikipedia.org/wiki/Injection_de_d%C3%A9pendance)

# Injection de dépendance

---

```
class Noel
{
    private $pereNoel;

    function __construct()
    {
        $this->pereNoel = new PereNoel();
    }
}
```

```
class Noel
{
    private $pereNoel;

    function __construct(PereNoel $pereNoel)
    {
        $this->pereNoel = $pereNoel;
    }
}
```



# Debug et Développement local

les bases pour une application  
Symfony

- les Logs
- VarDumper
- WebServer Local



# Les Logs

---

Les logs sont essentiels pour monitorer une application.

Symfony se base sur Monolog.

```
composer require monolog
```

# Les bons logs

---

Un bon log est un log :

- qui donne la criticité de ce qui s'est passé
- qui explique ce qui s'est passé
- qui donne le contexte dans lequel cela s'est passé



# Niveau de criticité d'un log

---

1. **debug** -> Detailed debug information
2. **info** -> Interesting events
3. **notice** -> Normal but significant events
4. **warning** -> Exceptional occurrences that are not errors
5. **error** -> Runtime errors that do not require immediate action but should typically be logged and monitored
6. **critical** -> Critical conditions
7. **alert** -> Action must be taken immediately
8. **emergency** -> System is unusable

# Niveau de criticité d'un log

---

1. vendu un pain au chocolat à 11h43 -> **debug**
2. vendu 46 pains au chocolat (largement au dessus de la moyenne) -> **info**
3. vendu 146 pains au chocolat (du jamais vu) -> **notice**
4. à court de pains au chocolcat à 8h45 -> **warning**
5. la machine qui fabrique les pains au chocolat est en panne -> **error**
6. les machines qui fabriquent les viennoiseries sont en panne -> **critical**
7. les portes de la boulangeries ne peuvent plus être ouvertes -> **alert**
8. la boulangerie est en feu -> **emergency**

# The VarDumper Component

---

`var_dump` de php est régulièrement utilisé pour déboguer.

Symfony a introduit une alternative à `var_dump` : le composant VarDumper  
`composer require --dev var-dumper`

- Mieux intégré à Symfony
- Plus facile à lire

⚠ à n'installer que pour l'environnement de développement

# Web Server

---

Pour avoir rapidement un serveur sans installer apache, il y a 3 possibilités :

- Symfony Local Web Server s'intègre parfaitement à Symfony Cloud (à utiliser de préférence avec Symfony5)
- PHP's built in Web Server via Symfony ~~est simple à utilisé~~ (a disparu avec Symfony5)
  - ~~composer require --dev symfony/web-server-bundle~~
  - ~~bin/console server:start~~
- PHP's built-in Web Server RAW s'adapte à tous les projets php
  - `php -S localhost:8000 -t public`

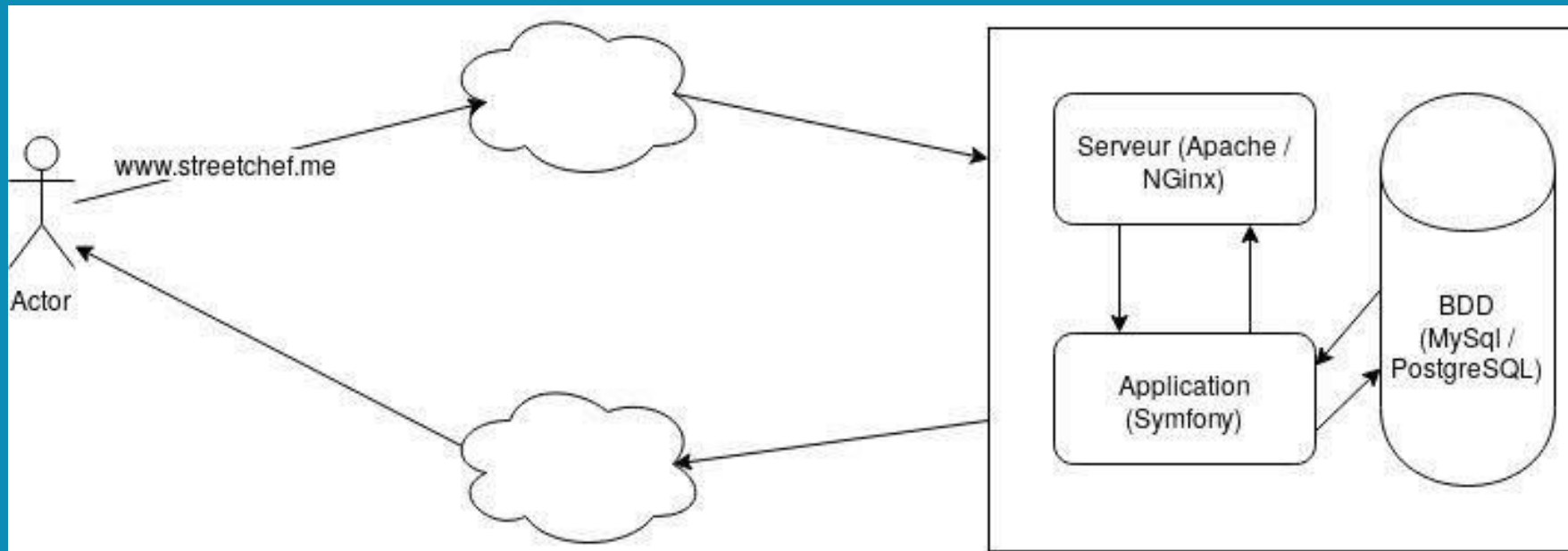
# Découpage d'une application Symfony

Les principales briques

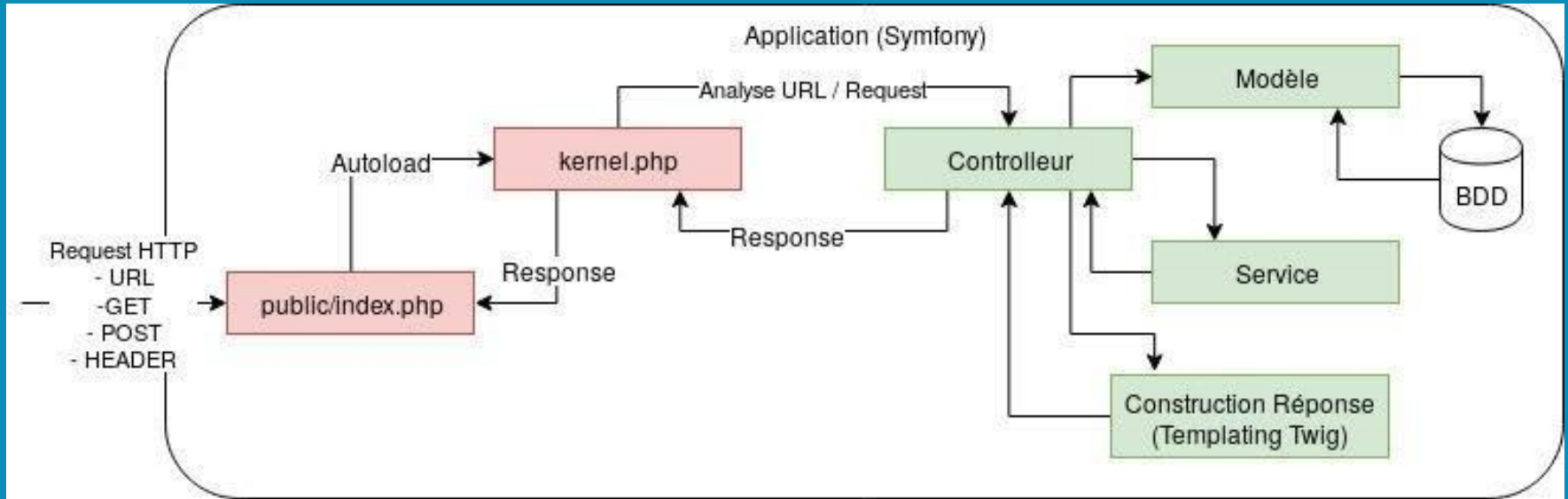
- Vue d'ensemble
- Controller
- Service
- Autowiring
- Commande
- Données



# Requêter une page web



# Dans l'Application Symfony



# Controller

---

- correspond à une classe dans le dossier `Controller` dont le nom est suffixé par `Controller`
- permet une correspondance url -> fonction
- on peut avoir de *jolies* url
- on peut rendre les url "dynamiques"
- La configuration en annotation de préférence (yaml, xml et php sont aussi possible)
- le bundle `doctrine/annotations` est requis pour les annotations

Doc : <https://symfony.com/doc/current/controller.html>



# Controller

```
localhost:8000/main  
{  
  message: "Welcome to your new controller!",  
  path: "src/Controller/MainController.php"  
}
```

```
namespace App\Controller;  
  
use ...  
  
class MainController extends AbstractController  
{  
    /**  
     * @Route("/main", name="main")  
     */  
    public function index()  
    {  
        return $this->json([  
            'message' => 'Welcome to your new controller!',  
            'path' => 'src/Controller/MainController.php',  
        ]);  
    }  
}
```

# Controller

---

- Annotation Route
  - Il faut ajouter `use Symfony\Component\Routing\Annotation\Route;`
  - Si ce n'est pas déjà fait : `composer require annotations`
- paramètres :
  - le chemin `"/chemin"`, avec entre `{ }` les variables du chemin (facultatif)
  - le nom du chemin `name="nom_du_chemin"`
  - facultatif : les regex que doivent respecter les variables `requirements={ "nom_var"="regex" }`
  - facultatif : les verbes HTTP autorisés `methods={ "GET", "POST" }`

Et beaucoup d'autres choses sont possibles (internationalisation, method, ...)

Doc : <https://symfony.com/doc/current/routing.html>

# Controller

```
localhost:8000/main/Carlos  
  
{  
  message: "Welcome to your new controller Carlos!",  
  path: "src/Controller/MainController.php"  
}
```

```
namespace App\Controller;  
  
use ...  
  
class MainController extends AbstractController  
{  
    /**  
     * @Route("/main/{name}", name="main", requirements={"name"="\w*"})  
     */  
    public function index($name = '')  
    {  
        return $this->json([  
            'message' => 'Welcome to your new controller '.$name.'!',  
            'path' => 'src/Controller/MainController.php',  
        ]);  
    }  
}
```

# Controller

---

Il est aussi possible de définir un préfixe de Route au niveau de la classe.

```
/**  
 * @Route("/prefix")  
 */
```

# Controller

---

```
localhost:8000/prefix/main/Carlos  
{  
  message: "Welcome to your new controller Carlos!",  
  path: "src/Controller/MainController.php"  
}
```

```
/**  
 * @Route("/prefix")  
 */  
class MainController extends AbstractController  
{  
    /**  
     * @Route("/main/{name}", name="main", requirements={"name"="\w*"})  
     */  
    public function index($name = '')
```

# Controller

---

Pour déboguer les routes :

```
bin/console debug:router
```

```
bin/console debug:router nom_route
```

```
bin/console router:match /path
```

# TD *Renvoyer la valeur binaire d'un nombre décimal*

---

# Service + Service Container

---

Un *service* est un objet au sens POO.

Dans l'exemple précédent, on peut considérer que `PereNoel` et `Noel` sont des objets.

Le service container va instancier ces objets, une et une seule fois, pour que nous puissions y accéder facilement.



# Service Container

---

Historiquement, on était obligé de définir *comment* instancier ces objets dans le fichier `services.yml` (renommer `services.yaml`).

Doc :

[https://symfony.com/doc/current/service\\_container.html](https://symfony.com/doc/current/service_container.html)

```
# app/config/services.yml
services:
    # ...

    # same as before
    AppBundle\:
        resource: '../src/AppBundle/*'
        exclude: '../src/AppBundle/{Entity,Repository}'

    # explicitly configure the service
    AppBundle\Updates\SiteUpdateManager:
        arguments:
            $adminEmail: 'manager@example.com'
```

# Autowiring

L'autowiring consiste pour symfony à se passer de la définition des services dans `services.yaml`, à instancier les services et à les injecter partout où ils sont nécessaire (Service / Controller / Command / ...).

Pour avoir la liste, exécuter :

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

Doc :

[https://symfony.com/doc/current/service\\_container/autowiring.html](https://symfony.com/doc/current/service_container/autowiring.html)

```
# config/services.yaml
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
        public: false       # Allows optimizing the container by removing unused services; this also means
                            # fetching services directly from the container via $container->get() won't work.
                            # The best practice is to be explicit about your dependencies anyway.

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App\:
        resource: '../src/*'
        exclude: '../src/{DependencyInjection,Entity,Migrations,Tests,Kernel.php}'
```

# Les Commandes dans Symfony

---

Une commande se lance depuis un terminal.

Pour voir la liste des commandes : `bin/console`

Ajouter une commande revient à :

- créer une classe dans le dossier `Command`
- dont le nom est suffixé par `Command`
- et qui étend `Command` de Symfony

Doc : <https://symfony.com/doc/current/console.html>

# Les Commandes dans Symfony

---

Les Commandes peuvent prendre des arguments et des options.

Exemple :

```
bin/console mail:send carlos@streetchef.me  
--subject="Bienvenue" --body="Vous cherchez un food truck  
?"
```

`carlos@streetchef.me` est un argument  
`--subject="Bienvenue"` est une option

# TD *Création d'une commande qui renvoie des blagues*

---

# TD *Création d'une commande qui renvoie des blagues 2 (joker inside)*

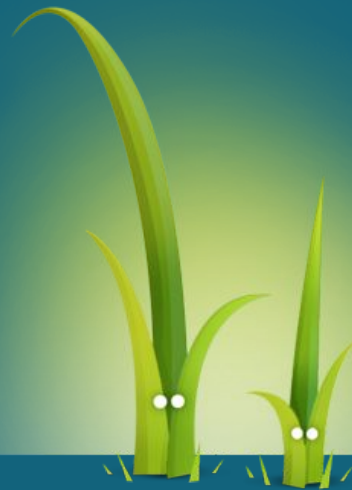
---

# TP *Quizz*

---

# Twig

The flexible, fast, and secure  
template engine for PHP





# Twig

---

- Un moteur de template sorti en 2009
- s'inspire de Jinja, un moteur de template pour Python
- Remplace PHP dans les fichiers de vue
- Créé par SensioLabs
- Facile à prendre en main
- Incontournable pour envoyer des emails

Documentation : <https://twig.symfony.com/doc/3.x/>

# Twig

---

Installation :

```
composer require twig
```

# Twig

---

## Conventions Twig :

- les templates sont dans le dossier `templates`
- extension `.html.twig` (si on veut générer un fichier html)

## Dans le cadre de fichiers twig pour les controller :

- un fichier Twig par méthode de controller
- les fichiers Twig sont nommés comme la méthode de controller associée
- on place les templates dans un sous-dossier nommé comme le contrôleur
- Pas de majuscules

# Twig

---

```
class MainController extends AbstractController
{
    /**
     * @Route("/main", name="main")
     */
    public function index()
    {
        $this->render( view: 'main/index.html.twig');
    }
}
```

# TD *Mon premier template* *Twig*

---

# les délimiteurs Twig

---

## Les Délimiteurs Twig

- Permettent de différencier le code Twig du reste du fichier
- Il y a toujours un délimiteur ouvrant et un fermant
- On peut en utiliser plusieurs dans un même fichier

# les délimiteurs Twig

---

## Écriture de commentaire en Twig

- Le délimiteur `{# ... #}` permet d'écrire des commentaires
- Commentaires invisibles dans le fichier final qui sera généré
- Équivalent à `<?php /** un commentaire */ ?>` en PHP

# les délimiteurs Twig

---

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Welcome!</title>
  </head>
  <body>
    {# Ceci est un commentaire twig. Il ne sera pas présent dans le fichier final. #}
  </body>
</html>
```



# les délimiteurs Twig

---

## Exécution de code dans un fichier Twig

- le délimiteur `{% ... %}` exécute du code avec une balise/tags à l'intérieur

Permet de :

- Créer des variables
- Réaliser une boucle
- Tester une condition
- Exécuter des fonctions
- ...
- Utilisé aussi délimiter des fichiers Twig

# les délimiteurs Twig

```
<body>
    {# on set la variable déjeuner #}
    {% set déjeuner = "foodtruck" %}

    {# on teste la variable #}
    {% if déjeuner == "foodtruck" %}

        {# on ne rentre ici que si la condition est vraie #}
        regardez sur https://www.streetchef.me

    {% endif %}
</body>
```

# les délimiteurs Twig

---

## Affichage de la valeur d'une variable en Twig

- `{{ ... }}` affiche quelque chose
- Équivalent au echo PHP : `<?php echo "du texte"; ?>`
- On y inclut habituellement une variable twig

# les délimiteurs Twig

```
{# on teste la variable #}
{% if dejeuner == "foodtruck" %}

    {# on ne rentre ici que si la condition est vraie
    regardez sur https://www.streetchef.me

    {# on affiche la valeur de dejeuner #}
    ✨ | pour trouver un {{ dejeuner }}

{% endif %}
```

# les balises Twig

---

Des balises (aussi appelé tags) sont intégrées à Twig nativement. Symfony étend twig et fournit des balises/tags supplémentaires.

Voir <https://twig.symfony.com/doc/3.x/>

Les principales balises sont :

- set
- if
- for

# les balises Twig

---

## Balise **set**

Doc : <https://twig.symfony.com/doc/3.x/tags/set.html>

- set crée une variable et affecte une valeur
- Pas de \$
- Les chaînes sont entre guillemets simples (ou doubles) : 'text'
- Les nombres sont saisis sans guillemets : 42
- Les booléens aussi : true

# les balises Twig

---

## Balise **set**

- Les tableaux sont entre crochets :

```
['rouge', 'vert', 'bleu']
```

- Les objets sont entre accolades :

```
{'film': 'Star Wars V', 'sortie': 1980, 'le_meilleur':  
true}
```

# les balises Twig

---

## Balise if

Doc : <https://twig.symfony.com/doc/3.x/tags/if.html>

- Permet d'exécuter des alternatives
- Se termine par `endif`
- Les opérateurs de comparaison habituels de php (`==`, `!=`, `>`, `<`, etc...) existent pour la plupart
- On utilise les mots-clefs `and` et `or`
- `else` et `elseif` pour créer des branches



# les balises Twig

---

## Balise **for**

<https://twig.symfony.com/doc/3.x/tags/for.html>

- `for` réalise une itération, une boucle
- Unique boucle en Twig

Permet de :

- Boucler sur un tableau ou un objet, avec ses clefs ou pas
- Boucler sur une liste de nombres ou de lettres
- Boucler en fonction de conditions
- Avoir accès à des informations sur la boucle en cours

# TD *Les balises Twig*

---

# Les filtres Twig

---

Les filtres twig peuvent :

- modifier une variable
- s'enchaîner
- avoir des options/arguments
- s'utiliser via un | (pipe) en suffixe

# Les filtres Twig

---

Des filtres sont intégrées à Twig nativement.  
Symfony étend twig et fournit des filtres supplémentaires.

Voir <https://twig.symfony.com/doc/3.x/>

On peut citer les filtres :

- `upper:Pereira | upper`
- `lower:Carlos | lower`
- `length:['shauri', '.', 'fr']|length`
- `raw:'<b>la balise html ne sera pas échappé</b>' | raw`
- `date:person.birthday | date("d/m/Y")`

# TD *Les filtres Twig*

---

# Gabarit Twig

---

Twig permet de créer des gabarits avec `{% block nom_du_block %}` définis à l'intérieur.

Et il est possible d'étendre ces gabarits et de ne redéfinir que les blocks.

- Les gabarits sont des *cadres* réutilisable
- Cela évite la répétition du code

# Gabarit Twig

---

Définition d'un gabarit dans le fichier

`base.html.twig`

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

# Gabarit Twig

---

On étend le gabarit et on redéfinit les blocks `title` et `body`.

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

{% block title %}nouveau titre de page{% endblock %}

{% block body %}
Nouveau contenu de body
{% endblock %}
```



# Gabarit Twig

---

Twig met aussi à disposition une fonction `include` comme celle de PHP.

```
{% include('template.html.twig') %}
```

Avantages :

- Les includes sont des *portions de page* réutilisable
- Cela évite la répétition du code

# TD *Les gabarits Twig*

---

# Les données du controller au Twig

---

Depuis le controller, le 2e argument de la fonction `render` est un tableau qui contient les données à passer au controller.

```
return $this->render( view: 'break_bad_template/index.html.twig', [
    'sayMyName' => 'Heisenberg',
]);
```

# Les données du controller au Twig

---

les clés du tableau sont utilisés pour définir les noms des variables dans le template twig.

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

{% block body %}
    You're {{ sayMyName }} !
{% endblock %}
```

# Les données du controller au Twig

Pro Tips :

La fonction php `compact` permet de créer facilement des tableaux contenant beaucoup de variables.

La clé dans le tableau sera le nom de la variable.

<https://www.php.net/manual/fr/function.compact.php>

```

        $sayMyName = 'Heisenberg';
        $profession = 'cook';

        return $this->render(
            view: 'break_bad_template/index.html.twig',
            compact( varname: 'sayMyName', _ : 'profession')
        );
```

# TD *Les Données du controller au Twig*

---

# Gestion des assets avec Twig

---

Les assets sont les images, css, js, documents qui sont versionnés avec projet.

Ces assets ont une url qui change selon l'environnement où on est (dev, preprod, prod) et la page où on se trouve.

La réécriture d'url à chaque génération de pages est nécessaire.

# Gestion des assets avec Twig

---

Le bundle asset permet de gérer ces problématiques.

Pour l'installer :

```
composer require asset
```

Doc : <https://symfony.com/doc/current/components/asset.html>



# Gestion des assets avec Twig

---

le bundle asset fournit une fonction twig `asset`.

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

Le fichier `css/style.css` se trouve dans le dossier `public`.

doc :

<https://symfony.com/doc/current/templates.html#linking-to-css-javascript-and-image-assets>

# Gestion des urls avec twig

---

Les urls changent selon l'environnement où on est (dev, preprod, prod).

La réécriture d'url à chaque génération de pages est nécessaire.

# Gestion des urls avec twig

---

Symfony nous fournit la fonction twig `path` qui nous renvoie une url relative  
`/inscription`

```
{{ path('main') }}
```

Symfony nous fournit aussi la fonction twig `url` qui nous renvoie une url absolue  
`https://www.streetchef.me/inscription`

```
{{ url('main') }}
```



# TP *Quizz Twig*

---

# TP *Coffre Fort*

---

# Formulaires

Récupération de données  
utilisateur

- Introduction et Installation
  - Création et affichage d'un formulaire
  - Traitement et enregistrement d'un formulaire
  - Sécurité
-

# Formulaires

---

Les formulaires permettent à une application de récupérer des données utilisateurs

C'est une tâche récurrente et fastidieuse dans une application



# Formulaire

---

Le composant `symfony/form` permet de :

- Automatiser les tâches
- Créer un html ultra simple prêt à être affiché
- d'hydrater automatiquement l'entité correspondante
- Se connecter naturellement au composant validator

# Installation

---

Installation :

```
composer require form
```

Doc : <https://symfony.com/doc/current/forms.html>

# Création d'un formulaire

---

Pour se simplifier la tâche, on va utiliser le `make bundle`  
`composer require maker`

et lancer la commande :

```
bin/console make:form
```

Une interface en ligne de commande va permettre de nommer le formulaire et le relier à une entité.

⚠ Le nom d'un formulaire est toujours suffixé de `Type`

`Maker` créera les formulaires dans le dossier `src/Form`

# Définition des champs

---

Pour chaque champ du formulaire, on peut définir :

- un type
- une liste option

# Formulaires

---

```
$builder
->add(
  child: 'nom',
  type: TextInputType::class,
  [
    'required' => true,
    'label' => 'nom'
  ]
);
```

# Types de champs

---

Il existe des dizaines de types de champs, les plus communs :

- Text
- TextArea
- Email
- Integer
- Date
- ...

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

# Options des types de champs

---

Chaque type de champ a ses propres options.

Les plus communes sont :

- label
- required
- attr

Doc : <https://symfony.com/doc/current/forms.html#form-type-options>

# Affichage du formulaire dans twig

---

Dans un Controller, il faut :

1. Créer une instance de l'entité
2. Créer une instance du formulaire
3. Passer le formulaire à Twig



# Affichage du formulaire dans twig

---

```
class PersonController extends AbstractController
{
    /**
     * @Route("/person", name="apprenti")
     */
    public function index()
    {
        $person = new Person();
        $personForm = $this->createForm( type: PersonType::class, $person);

        return $this->render(
            view: "Person/index.html.twig",
            [
                "personForm" => $personForm->createView()
            ]
        );
    }
}
```

# Affichage du formulaire dans twig

---

Dans Twig, il faut :

1. Afficher le formulaire

# Affichage du formulaire dans twig

---

```
<div>
    {# affichage simple #}
    {{ form(personForm) }}
</div>
```

# Affichage du formulaire dans twig

---

```
<div>
    {# décomposition du formulaire #}
    {{ form_start(personForm) }}

    {{ form_widget(personForm) }}

    <button type="submit">Envoyer</button>

    {{ form_end(personForm) }}
</div>
```

# Affichage du formulaire dans twig

---

```
1 <div>
2     {# décomposition par ligne #}
3     {{ form_start(personForm) }}
4
5     {{ form_row(personForm.firstName) }}
6     {{ form_row(personForm.lastName) }}
7     {{ form_row(personForm.birthday) }}
8
9     <button type="submit">Envoyer</button>
10
11     {{ form_end(personForm) }}
12 </div>
```

# Affichage du formulaire dans twig

---

```
{%><div>
  {%->{# décomposition par ligne #}
  {%->{{ form_start(personForm) }}

  {%->{{ form_label(personForm.firstName) }}
  {%->{{ form_widget(personForm.firstName) }}
  {%->{{ form_errors(personForm.firstName) }}

  {%->{{ form_label(personForm.lastName) }}
  {%->{{ form_widget(personForm.lastName) }}
  {%->{{ form_errors(personForm.lastName) }}

  {%->{{ form_label(personForm.birthday) }}
  {%->{{ form_widget(personForm.birthday) }}
  {%->{{ form_errors(personForm.birthday) }}

  {%-><button type="submit">Envoyer</button>

  {%->{{ form_end(personForm) }}
{%></div>
```

# *TD TV Show*

---

# Traiter le formulaire

---

Le traitement du formulaire se fait dans le controller, sur la même page.  
Pour traiter le formulaire, il faut :

- s'assurer que le formulaire a été soumis
- injecter les données dans l'entité
- traiter les données
- rediriger l'utilisateur vers une autre route ou lui afficher un message



# Traiter le formulaire

---

```
/**
 * @Route("/person", name="apprenti")
 */
public function index(Request $request)
{
    $person = new Person();
    $personForm = $this->createForm('type: PersonType::class', $person);

    $personForm->handleRequest($request);
    if ($personForm->isSubmitted()) {
        // on fait quelque chose des données
        var_dump($person);

        return $this->redirectToRoute('route: "done");
    }

    return $this->render(
        view: "Person/index.html.twig",
        [
            "personForm" => $personForm->createView()
        ]
    );
}
```

# Sécuriser le formulaire

---

Qu'est-ce qu'une attaque CSRF :

- Soumission d'un formulaire à notre insu sur un site sur lequel on est connecté

Protection contre les attaques CSRF :

- Ajout d'un champ caché dont seul l'application connaît la valeur

<https://www.cert.ssi.gouv.fr/information/CERTA-2008-INF-003/>

# Sécuriser le formulaire

---

Symfony met à disposition un système de sécurisation des formulaires contre les attaques CSRF.

```
composer require security-csrf
```

et il faut valider le formulaire.

# Sécuriser le formulaire

---

```
if ($personForm->isSubmitted() && $personForm->isValid()) {
```

# *TD TV Show 2*

---



**ANY QUESTIONS?**



# REST



REpresentational State Transfer



# REST

---

Défini par Roy Fieldling 2000 dans une thèse de doctorat.

- Permet de créer des services web (a.k.a API)
- Permet l'interopérabilité entre différents acteurs
- Permet d'accéder à des "Ressources" (correspond à un Objet)
- Stateless : La requête transporte toutes les informations nécessaires, notamment l'authentification

source : [wikipedia](#)



# REST

---

- Basé sur HTTP
  - verbes (aka méthodes)
    - GET
    - POST
    - PUT
    - PATCH
    - DELETE
    - ...

source : [wikipedia](#)

# REST

---

Utilisation de chaque verbe HTTP en REST :

- GET : pour récupérer les informations d'une resource
- POST : pour créer une resource dont on n'a pas l'identifiant. (éventuellement pour mettre à jour)
- PUT : pour mettre à jour une resource dont on a l'identifiant (éventuellement pour créer)
- PATCH : pour mettre à jour partiellement une resource
- DELETE : pour supprimer une resource

# REST

---

- Basé sur HTTP
  - Status Code
    - 200 : OK
    - 400 : Bad Request
    - 404 : Not Found
    - 500 : Internal Server Error
    - ...

source : [wikipedia](#)

# REST - Les concurrents

---

- SOAP
  - développé par Microsoft et IBM
  - normalisé par le W3C
  - basé sur le XML et une définition précise des interfaces
  - trop verbeux et trop contraignant
  - beaucoup utilisé AVANT REST
- GraphQL
  - développé par Facebook
  - Permet de requêter simplement uniquement les données nécessaires
  - très à la mode
  - toutes les requêtes se font en POST...

# Glory of REST

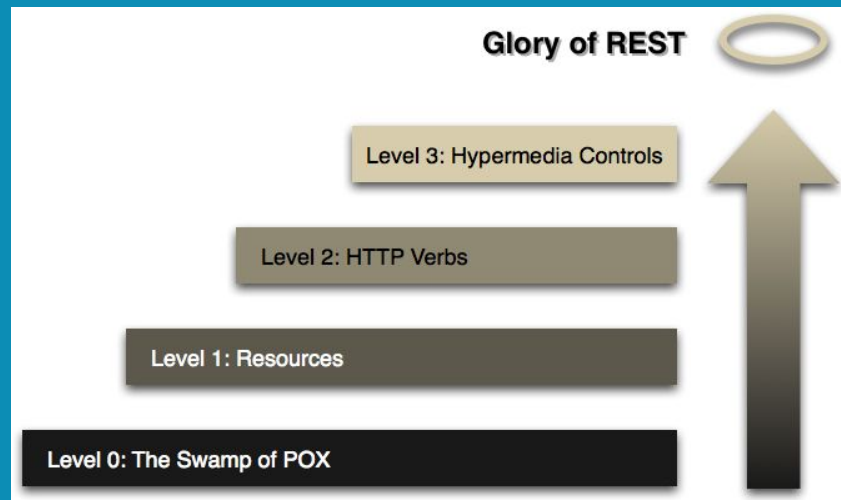
niveau 0 - On fait ce qu'on veut (et probablement n'importe quoi)

niveau 1 - On définit les ressources (objets) qu'on expose

niveau 2 - On expose ces objets en suivant les principes des verbes HTTP  
niveau 3 - on renvoie des liens pour accéder aux ressources associées

source :

<https://martinfowler.com/articles/richardsonMaturityModel.html>



# Rest et les liens

---

Dernière étape avant d'atteindre *The Gory of REST*, l'ajout de liens permet de naviguer entre les ressources.

Il existe différents standard pour définir des liens :

- [rfc8288](#) pour les headers HTTP
- [HATEOAS](#) le plus connu pour REST
- [HAL](#) pour Hypertext Application Language qui est encore en brouillon
- [JSON Hyper-Schema](#) qui est encore en brouillon
- ...

Il n'y a donc aucun standard réellement en place.

# TD *Rest - Level 0*

---

# TD *Rest - Level 1*

---



# TD *Rest - Level 2*

---

# TD *Rest - Level 3*

---



*So go ahead, ask me anything*

# Validation

---

# Pourquoi valider des données

---

La validation des données reçues permet de s'assurer que l'utilisateur envoie les données telles qu'attendues.

Ex :

- un email doit être bien formaté
- la taille d'un humain doit être positif et avec une limite
- ...

C'est une tâche commune à travers une application, et souvent fastidieuse.

# Validations des données avec Symfony

---

Symfony permet de :

- spécifier des contraintes pour les attributs d'un objet (souvent une entité)
- configuration en annotations (XML, YAML et PHP aussi possible)
- valider que ces contraintes sont respectées

Grâce au bundle Validator:

- `composer require validator`

# Contraintes

---

Dans le bundle Validator :

- Les règles de validation sont appelées `Constraint`
- Ces contraintes ont historiquement l'alias `Assert`
- On peut appliquer 0, 1, \* contraintes sur un attribut
- Il existe 40 contraintes de bases
- On peut créer nos propres contraintes
- Chaque contrainte accepte des paramètres précis

# Contraintes

---

Voici une liste des contraintes les plus utilisées :

- `NotBlank`
- `NotNull`
- `Email`
- `Url`
- `Length`



# Contraintes

---

Voici une liste des contraintes les plus utilisées :

- Regex
- Luhn
- Iban
- Isbn

Voir la liste complète

<https://symfony.com/doc/current/validation.html#supported-constraints>

# Exemple de validation

---

```
// src/Entity/Author.php
```

```
// ...
```

```
use Symfony\Component\Validator\Constraints as Assert;
```

# Exemple de validation

---

```
class Author
{
    /**
     * @Assert\NotBlank
     * @Assert\Length(min=3)
     */
    private $firstName;
}
```

# Validation

---

```
public function author(ValidatorInterface $validator)
{
    $author = new Author();

    // ... do something to the $author object

    $errors = $validator->validate($author);

    // ... do something else
}
```

# TD *Validation des données reçues*

---



I'LL TAKE YOUR QUESTIONS NOW.

# TP *Les Bibliothèques du Vatican*

---



**ASK ME ANYTHING.**



# Les Données

la mine d'or des applications

- Introduction
- Installation de Doctrine
- Génération de la Base De Données
- Requêtage basique de la Base De Données
- Requêtage avancé de la Base De Données
- Relations entre entités



# Les Données

---

- Besoin de la stocker
- Besoin de la structurer
- Besoin d'y accéder facilement et rapidement

-> Utilisation d'une Base De Données

# Bases De Données

---

- Les BDD relationnelles les plus connus
  - MySQL / MariaDB
  - PostgreSQL
  - Oracle
  - Sqlite

Les BDD relationnelles sont basées sur le SQL... avec quelques différences.  
Requêter une BDD revient à récupérer un tableau de données.

# Doctrine

---

Doctrine va nous permettre d'abstraire la Base De Données et de travailler avec des Objets au lieu de tableau.

# Doctrine

---

## Avantages :

- connaitre Doctrine, c'est être capable de travailler avec n'importe quelle BDD
- Dans un développement Objet, il est plus facile d'accéder à la BDD via des Objets

# Doctrine

---

## Inconvénients :

- Certaines requêtes complexes sont intrinsèquement impossible à faire avec Doctrine
- Pas du tout adapté au traitement d'un grand volume de données, car long et très consommateur de mémoire

# Installation

---

```
composer require doctrine
```

De nouveaux dossier sont apparus :

- `src/Entity`
- `src/Migrations`
- `src/Repository`

# Configuration

---

Ainsi qu'une configuration supplémentaire :

- dans le fichier d'environnement `.env`
- dans le dossier de configuration `config/packages/doctrine.yaml`

La configuration de l'accès à la base de données se fait dans le fichier `.env.local` en reprenant les variables définies dans `.env`

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
```



# Configuration

---

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
```

il faut remplacer :

- **db\_user** -> mettre votre login mysql (souvent root en local)
- **db\_password** -> mettre votre mot de passe mysql (souvent vide avec WAMP)
- pour un développement local, **127.0.0.1:3306** peut fonctionner
- **db\_name** -> mettre un nom de base de données. Par exemple **symfony** pour la formation

# Création de la base de données

---

Pour créer la base de données

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

⚠ Dans MySQL, le mot "schema" est utilisé pour parler de base de données

# Création d'une entité

---

Une entité est une classe dans le dossier `src/Entity`

- Grossièrement, on peut dire qu'une table correspond à un objet (appelé une entité)
- Chaque ligne de la table correspond à une instance de l'objet
- Chaque colonne de la table correspond à un attribut (mappé) de l'objet

Un mapping est fait entre chaque table, et ses colonnes, de la base de données et un objet, et ses attributs.

Doc : <https://symfony.com/doc/current/doctrine.html#creating-an-entity-class>

# Création d'une entité

```
<?php

namespace App\Entity;

class Person
{
    private $id;

    private $firstName;

    private $lastName;

    private $birthday;
```

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

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

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

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

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $birthday;
```

# Création d'une entité

---

Pour se simplifier la tâche, on va utiliser le `make bundle`

`composer require maker`

et lancer la commande :

`bin/console make:entity`

Une interface en ligne de commande va permettre de configurer le mapping de l'entité.

# Création d'une entité

---

Cela va générer automatiquement :

- l'entité dans le dossier `src/Entity`, avec ses attributs et ses getters/setters
- le fichier repository dans le dossier `src/Repository`, avec des exemples de fonctions

# Synchronisation de la base de données

---

## La bonne pratique

Création d'un fichier de migration de base de données grâce au maker bundle

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

Le fichier est généré dans `src/Migrations`

Le nom du fichier est au format

```
Version<année><mois><jour><heure><minute><seconde>.php
```

# Synchronisation de la base de données

---

Vérifier le contenu du fichier et exécutez la migration

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

Une table `migration_versions` est créée la première fois pour historiser les migrations effectuées.

Doc :

<https://symfony.com/doc/master/bundles/DoctrineMigrationsBundle/index.html>



# Synchronisation de la base de données

---

## La pratique la plus rapide

Il existe une commande pour forcer la mise à jour de la base de données

```
bin/console doctrine:schema:update --force
```

et on marque toutes les migrations comme jouée

```
bin/console doctrine:migrations:version --add --all
```



À ne faire que si vous *maitrisez* ce que vous faites

# TD *L'école*

---

# Requêtage basique

---

Pour accéder à la base de données, il faut passer par l'**Entity Manager**

Pour cela, on peut injecter le service `EntityManagerInterface`

💡 Symfony nous donne un autre accès via son `AbstractController` :

```
$entityManager = $this->getDoctrine()->getManager();
```

# Requêtage basique

```
<?php

namespace App\Controller;

use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class PersonController extends AbstractController
{
    /**
     * @Route("/person", name="person")
     */
    public function index(EntityManagerInterface $entityManager)
    {
```

```
<?php

namespace App\Controller;

use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class PersonController extends AbstractController
{
    /**
     * @Route("/person", name="person")
     */
    public function index()
    {
        $entityManager = $this->getDoctrine()->getManager();
```

# Requêtage basique

---

Pour ajouter une entrée en base de données ou en modifier une existante

```
$entityManager->persist($object);
```

Pour supprimer une entrée en base de données

```
$entityManager->remove($object);
```

`persist` et `remove` font la modification dans la mémoire interne de doctrine.

Pour que cela impacte la base de données, il faut "flusher"

```
$entityManager->flush();
```

# Requêtage basique

---

```
/**
 * @Route("/person", name="person")
 */
public function index(EntityManagerInterface $entityManager)
{
    // on crée une nouvelle instance de Person
    $person = new Person();
    $person->setLastName( lastName: 'Bond')
    $person->setFirstName( firstName: 'James')
    $entityManager->persist($person); // on indique à Doctrine son existence
    $entityManager->flush(); // on l'enregistre en BDD

    $person->setBirthDay(new \DateTime( time: '1982-12-13'));
    $entityManager->persist($person); // on indique à Doctrine qu'on a modifié l'objet
    $entityManager->flush(); // on l'enregistre en BDD

    $entityManager->remove($person); // on indique à Doctrine que cette personne n'existe plus
    $entityManager->flush(); // on l'enregistre en BDD
}
```

# Requêtage avancé

---

Le requêtage avancé passe par la récupération du *Repository* associé à l'entité.

La récupération du *Repository* passe par l'Entity Manager.

```
$repository = $entityManager->getRepository(Object::class)
```



Symfony nous donne un autre accès via son `AbstractController`:

```
$repository =  
$this->getDoctrine()->getRepository(Object::class)
```

Il y a un repository par entité, défini en haut de la classe de l'entité.

# Requêtage avancé

---

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonRepository")
 */
class Person
```



# Requêtage avancé

---

```
/**
 * @Route("/person", name="person")
 */
public function index(EntityManagerInterface $entityManager)
{
    $repository = $entityManager->getRepository( className: Person::class);

    $repository = $this->getDoctrine()->getRepository( persistentObject: Person::class); // fait exactement la même chose
```

# Requêtage avancé

---

```
$repository->findAll(); // récupère tous les objets
$repository->find($id); // récupère l'objet qui a pour
    identifiant $id
$repository->findOneByLastName($lastName); // DYNAMIQUE,
    récupère le premier objet trouvé qui a pour nom de famille
    $lastName
$repository->findOneBy(["lastName" => $lastName]); //
    pareil qu'au dessus
```

# Requêtage avancé

---

```
$repository->findByLastName($lastName, ["firstName" =>  
"DESC"]); // DYNAMIQUE, renvoie TOUS les objets dont le nom  
de famille est $lastName, trié par prénom par ordre  
alphabétique inversé  
$repository->count(); // renvoie le nombre d'élément dans  
la table
```

# Requêtage avancé

---

On peut aussi créer des requêtes plus complexe grâce à 2 "langages" :

- DQL
- Query Builder

# Requêtage avancé

Le DQL ressemble aux SQL, mais se base sur les données de l'objet.

De préférence, les requêtes seront écrites dans le repository.

```
/**
 * @param $minBirthday
 *
 * @return Person[] Returns an array of Person objects
 */
public function findByBirthdayDQL($minBirthday)
{
    $dql = "SELECT p
    FROM App\Entity\Person p
    WHERE p.birthday >= :birthday
    ORDER BY p.id DESC";

    $query = $this->getEntityManager()->createQuery($dql);
    $query->setParameter( key: 'birthday', $minBirthday);
    $query->setMaxResults( maxResults: 10);

    return $query->getResult();
}
```

# Requêtage avancé

---

Le Query Builder est un ensemble de classes et méthodes qui va aussi générer du DQL.

Cela permet de faire exactement la même chose que le DQL.

Pour la construction de certaines requêtes complexes, il peut être plus simple de travailler avec le Query Builder.

# Requêtage avancé

```
/**
 * @param $minBirthday
 * @return Person[] Returns an array of Person objects
 */
public function findByBirthdayDQL($minBirthday = null)
{
    $dql = "SELECT p
    FROM App\Entity\Person p";

    if ($minBirthday) {
        $dql .= "WHERE p.birthday >= :birthday";
    }

    $dql .= "ORDER BY p.id DESC";

    $query = $this->getEntityManager()->createQuery($dql);

    if ($minBirthday) {
        $query->setParameter( key: 'birthday', $minBirthday);
    }

    $query->setMaxResults( maxResults: 10);

    return $query->getResult();
}
```

```
/**
 * @return Person[] Returns an array of Person objects
 */
public function findByBirthdayQB($minBirthday = null)
{
    $querybuilder = $this->createQueryBuilder( alias: 'p');

    if ($minBirthday) {
        $querybuilder
            ->andWhere('p.birthday >= :val')
            ->setParameter( key: 'val', $minBirthday);
    }

    $query = $querybuilder
        ->orderBy( sort: 'p.id', order: 'DESC')
        ->setMaxResults( maxResults: 10)
        ->getQuery();

    return $query->getResult();
}
```

# TD *Les apprentis*

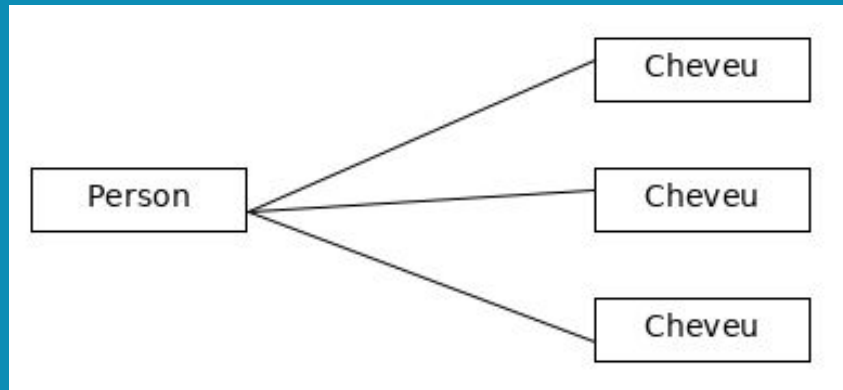
---



# Relations entre entités

---

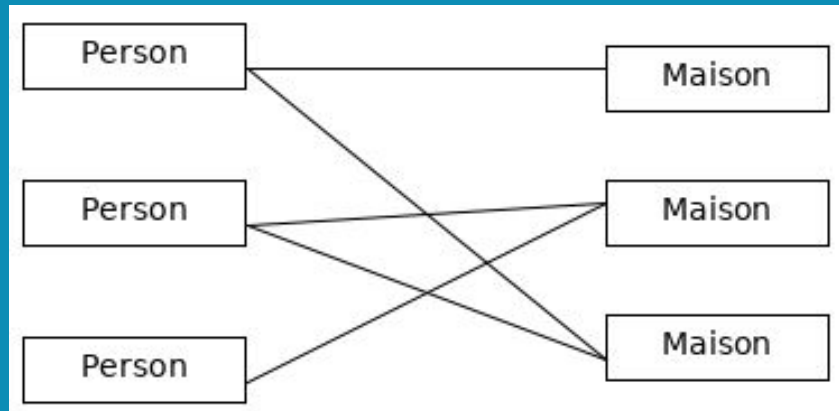
## Relation OneToMany / ManyToOne



# Relations entre entités

---

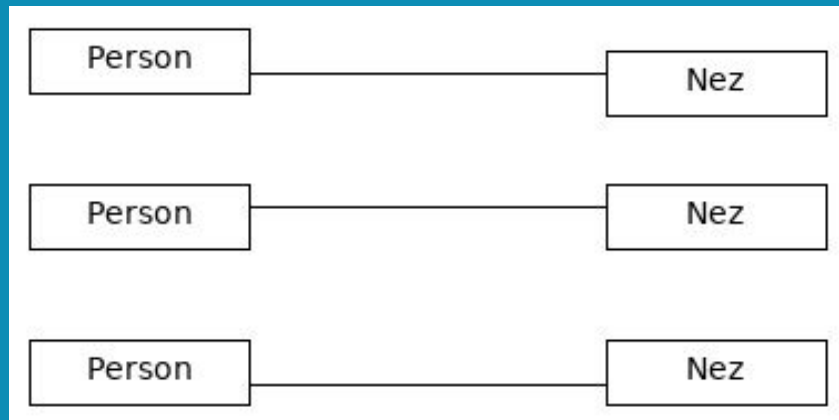
## Relations ManyToMany



# Relations entre entités

---

## Relations OneToOne



# Relations entre entités

---

Chaque relation est représentée par :

- un attribut dans chaque entité de la relation
- des annotations dans les commentaires au dessus de l'attribut

# Relations entre entités

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

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

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

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

    /**
     * @ORM\OneToOne(targetEntity="App\Entity\Person", inversedBy="nez", cascade={"persist", "remove"})
     * @ORM\JoinColumn(nullable=false)
     */
    private $person;
```

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

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

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

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $birthday;

    /**
     * @ORM\OneToOne(targetEntity="App\Entity\Nez", mappedBy="person", cascade={"persist", "remove"})
     */
    private $nez;
```

# Relations entre entités

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

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

    /**
     * @ORM\ManyToMany(targetEntity="App\Entity\Person", inversedBy="maisons")
     */
    private $persons;
```

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

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

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

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $birthday;

    /**
     * @ORM\ManyToMany(targetEntity="App\Entity\Maison", mappedBy="persons")
     */
    private $maisons;
```

# Relations entre entités

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

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

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Person", inversedBy="cheveus")
     * @ORM\JoinColumn(nullable=false)
     */
    private $person;
```

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

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

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

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $birthday;

    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Cheveu", mappedBy="person", orphanRemoval=true)
     */
    private $cheveus;
```

# Relations entre entités

---

Doctrine hydrate *automatiquement* les objets liés en *lazy loading* (uniquement lorsque nécessaire).

- ⚠ Le Lazy Loading peut provoquer énormément de requêtes
- ⚠ Le Lazy Loading est impossible dans une relation OneToOne

Doctrine hydrate *automatiquement* les identifiants (\$id) à l'insertion en base de chaque objet.



# Relations entre entités

---

*Cascade persist* signifie qu'à l'insertion de *Person* en BDD, les objets *Nez* associés seront aussi insérés

*Cascade remove* signifie qu'à la suppression de *Person* de la BDD, les objets *Nez* associés seront eux aussi supprimés

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Nez", mappedBy="person", cascade={"persist", "remove"})
 */
private $nez;
```

# TD *Formateur*

---



***Come on, ask questions.  
This is so awkward.***

# TP *Les bases des Bibliothèques du Vatican*

---



**You can ask me anything you want,  
ask me anything.**

