

Formation Symfony

1. Rappel / Pré-requis

Le code associé à ce cours est disponible sur github :

<https://github.com/ice-devel/formation-sf-axalone.git>

a. POO Programmation Orientée Objet

La POO est possible en PHP depuis la version 3. Il s'agit d'une structuration différente du code, en opposition à la programmation procédurale.

Le but est de modéliser des objets qui représenteront les concepts métiers de notre application. On nomme ces modélisations des classes. Un objet est créé à partir d'une classe : il est une instance de classe.

Par exemple, vous développez une boutique e-commerce, vos classes peuvent être les "Produits", on encore les "Commandes", et ces classes peuvent avoir des relations entre elles. Dans l'exemple ici, une commande peut contenir un ou plusieurs produits.

En modélisant ainsi le code, on va centraliser toutes les propriétés et les fonctions d'un même concept à un seul endroit. Les classes se déclare avec le mot-clé "class", suivi des propriétés et des méthodes dont il faudra définir la visibilité (private, protected, public). Parmi les méthodes, certaines reviendront régulièrement pour tout objet : les accesseurs/mutateurs (getters/setters), le constructeur, ou encore certaines méthodes dites magiques comme le toString.

A ce sujet, une bonne pratique est de respecter le principe d'encapsulation.

Voir le code : [Produit.php](#) et [Commande.php](#)

b. POO avancée : héritage / interface / trait

- Héritage

Dans la POO, il existe des techniques avancées puissantes comme l'héritage. Si plusieurs classes se réfèrent au même concept métier, elles pourraient partager des propriétés et/ou des méthodes sans avoir à les définir pour chacune d'entre-elles.

Imaginons par exemple qu'une application gèrent les employés d'une entreprise. Chacun a ses spécificités : le chef de projet peut être en relation avec les clients, le développeur va coder, le graphique va créer un design, mais pourtant ils ont tous des choses en commun : un nom, un prénom, un salaire, etc.

On peut donc créer une classe "Employee" qui va centraliser ces propriétés et méthodes communes, et une classe par poste comme les classes "Developper" et "Designer", qui hériteront de la classe "Employee".

On parlera de classe "mère" et de classe "fille". Une mère peut avoir plusieurs filles, mais une fille ne peut avoir qu'une seule mère. Si une classe B hérite d'une classe A, on peut alors dire que B est un A, mais pas l'inverse. Dans notre exemple, un développeur est un employé, mais un employé n'est pas forcément un développeur.

Voir le code et ses commentaires : [Employee.php](#) et [Developper.php](#)

Un héritage peut également se faire sur plusieurs niveaux : une fille peut à son tour être une mère, s'il y a besoin de spécifier davantage les propriétés ou les méthodes. En reprenant l'exemple ci-dessus, si nous avons besoin de séparer une logique métier différente pour les développeurs, on peut leur créer un nouveau niveau. Le développeur PHP code du PHP, alors que le développeur Java code du Java.

Par ailleurs, dans les classes filles, les méthodes sont héritées, ce qui veut dire que lorsque qu'une fille appelle une fonction de la classe mère, c'est le code se trouvant dans cette dernière qui est exécutée. Si la classe fille a besoin de cette fonction mais requiert une spécificité, elle peut alors redéfinir cette méthode pour y changer le code. La visibilité de la fonction surchargée doit cependant être égal ou plus faible que la visibilité originale. Dans notre exemple, imaginons qu'un développeur code en binaire par défaut, mais que chaque développeur spécialisé doit coder dans son langage : les développeurs spécialisés redéfiniront cette méthode. Tous les non-spécialisés continueront avec le binaire.

[Voir le code et ses commentaires : *DevelopperPHP.php* et *DevelopperJava.php*](#)

- *Classes abstraites et finales*

Une classe abstraite est une classe qu'on ne peut instancier. Elle contient donc des définitions de propriétés et de méthodes, servant de modèle pour des classes filles.

Pourquoi rendre abstraite une classe ? Et bien simplement pour centraliser des propriétés/méthodes au même endroit, qui n'ont pas lieu d'exister sans une spécialisation. Dans le cas d'un développement modulaire, où plusieurs développeurs vont intervenir, la classe abstraite permettra aussi d'empêcher les autres développeurs d'utiliser directement cette classe alors que ça n'a pas de sens au sein de l'application.

Dans notre exemple, on peut très imaginer qu'un employé en tant que tel ne peut exister dans le programme. C'est forcément un graphiste, un développeur, etc.

[Voir le code et ses commentaires : *Employee.php*](#)

A l'inverse, une classe finale est une classe qui ne peut être héritée. C'est le dernier chaînon de la hiérarchie. On peut rendre une classe finale si on a prévu que notre application ne pourrait fonctionner avec une fille de cette classe. Dans le cas d'un projet avec plusieurs développeurs, ou encore dans le cadre d'une bibliothèque distribuée sur le web à d'autres dévs, cela pourra empêcher les autres personnes d'utiliser cette classe comme une classe mère.

- *Méthodes abstraites et finales*

De la même manière que pour les classes, les méthodes de classe peuvent être abstraites ou finales.

Une méthode abstraite ne définit que la signature de la fonction, et toute classe fille devra obligatoirement redéfinir cette méthode pour y ajouter le corps.

A l'inverse, une méthode finale ne pourra pas être redéfinie au sein d'une classe fille.

- *Interface*

Une interface est composée de méthodes où seule la signature est définie. Pour qu'elle soit utilisée, une classe doit l'implémenter : cette classe devra redéfinir toutes les méthodes présentes dans l'interface. On implémente une interface avec le mot-clé "implements".

Contrairement à l'héritage, une classe peut implémenter une multitude d'interfaces. On implémentera toutes les interfaces en les séparant par des virgules.

Pour simplifier, une interface est donc un modèle sur lequel les classes doivent se baser.

Une interface est donc souvent prévue en amont d'un développement, afin que le code classes respecte les restrictions définies. A nouveau, dans un développement modulaire, cela permettra à tous les développeurs travaillant sur le projet de ne rien oublier.

Une interface permet aussi de typer des variables, pour s'assurer que ces variables auront les méthodes nécessaires dans la suite du code. Imaginons dans notre exemple que seuls les développeurs et les admins réseaux puissent rebooter un serveur. Nous pouvons créer une interface "ServerAdminInterface", qui oblige l'implémentation d'une méthode "reboot" avec en paramètre, pourquoi pas, le serveur à redémarrer.

Nous avons un service qui propose une fonction pour redémarrer un serveur : elle prend en paramètre un employé qui a la capacité de le faire. Sans l'interface, comment typer ce paramètre, en sachant que ça peut donc être un admin ou un développeur ? Nous serions obligés de créer deux paramètres : un pour le développeur et un autre pour l'admin, ce qui alourdira le code et ne facilitera pas sa compréhension.

Alors qu'avec l'interface, il suffira de typer cet unique paramètre, qui contient un employé avec la capacité de reboot. de cette manière : `ServerAdminInterface $employee`

Ainsi dans la suite de code de ce service, nous savons que cet employé pourra appeler la méthode "reboot", sans se soucier des autres méthodes, spécifiques ou non, que tout employé possède.

Voir le code et ses commentaires : [Developer.php](#), [Admin.php](#), [ServerAdminInterface.php](#), [ServerService.php](#)

- *Trait*

Code réutilisable dans des classes indépendantes : mettre l'alarme : [CEO](#), [Technicien](#), [Chef de service](#)

c. Architecture MVC

Une architecture MVC permet de mieux organiser son code, c'est aujourd'hui une pratique font on peut se passer, même si d'autres composantes structurelles entrent en jeu.

Il s'agit définir des rôles en séparant différentes logiques :

- **Models** : les entités représentant les données à gérer. Elles seront souvent mappées à une base de données.
- **Views** : les vues centralisant ce qui doit être affiché à l'utilisateur. Dans le cas d'un site web, c'est par exemple le html/css.
- **Controllers** : les contrôleurs orchestrent l'ensemble des opérations. Ils font le lien entre les entités et les vues. Les contrôleurs doivent être léger : les logiques métiers n'y ont pas leur place. Dans l'idéal, il faut qu'au premier coup d'œil on puisse voir et comprendre ce qu'il s'y passe, sans en avoir les détails.

Voir le code et ses commentaires : [controller.php](#), [view.php](#), [User.php](#)

d. Design Pattern

Les design patterns sont nés de problématiques de conception récurrentes.

Afin de répondre efficacement et de façon évolutive à certains problèmes, on peut évoquer :

- Pattern Singleton
- Pattern Strategy
- Pattern Factory
- Pattern Observer
- Injection de dépendances

2. Symfony

a. Historique du framework Symfony

b. Familiarisation avec le versionning et les gestionnaires de dépendance

- *Versionning : Git*
- *Gestionnaire de dépendances : Composer*

c. Installation de l'environnement Symfony

- Installation serveur web et bdd
- Installation symfony
- Installation composer
- Commande pour créer un projet symfony
 - `symfony new nom_projet` (projet réduit, pour api par exemple)
 - `symfony new nom_projet --full` (projet site web, avec système de template inclus par exemple)

d. Création / Structure / Configuration d'un projet

- Création d'un projet
- Structure
 - Bin
 - Config
 - Public
 - Src
 - Templates
 - Tests
 - Translations
 - Var
 - Vendors
 - Composer.json
 - .env
- Configuration bdd / .env
- Commandes

3. Controllers

a. Echange/client serveur façon Symfony : Request et Response

b. Le routing - Création d'une URL

- Route / pattern
 - Paramètre de route
 - Paramètre optionnel
 - Format de paramètre
- Générer un controller et une route :
 - `php bin/console make:controller`
- Pour créer une page, il faut donc :
 - Une route

- Une action de controller (fonction)
- Un template

c. Orchestrer les différents composants dans un controller

4. Views

a. Twig: utilisation d'un moteur de template

b. Générer un template

- Afficher {{ }}
- Faire {% %}
- Commentaire {# #}

c. Variables, structures conditionnelles et itératives dans les templates

- {% for value in array %} {%endfor %}
- {% if condition %} {%endif %}

d. bloc et héritage dans les templates

- {% block nom_bloc %}

e. include / render

On peut dans un template inclure un autre template avec include, et même appeler un controller avec render.

f. Travaux pratiques

Créer une page pour afficher une liste d'utilisateurs :

- Contrôler un nouveau controller UserController avec une route /users/{integer}
- Créer dans ce controller un tableau avec {integer} users aléatoire ['id', 'nom', 'enabled']
- Afficher dans un template tous les utilisateurs dans un table html, en mettant en rouge les lignes des utilisateurs dont le compte n'est pas activé
- Rajouter un bouton/liens pour chaque user : Supprimer

5. Entities

a. La couche métier

b. Principe d'un ORM : Doctrine et le mapping

Exit les requêtes SQL, on modélise nos entités, et on va laisser Doctrine gérer les interactions avec la BDD la majeure partie du temps.

La première étape consiste à configurer la chaîne de connexion à la base de données dans les variables d'environnement.

- Soit dans le fichier .env à la racine du projet
- Soit directement dans le vhost du serveur web (Apache, nginx, etc.)

c. Créer / modifier une entité

On peut évidemment créer notre entité à la main, ou utiliser une commande bien pratique :

php bin/console make:entity

d. Création de la base

Idem à la main (phpmyadmin, etc.) ou en ligne de commande :

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

On peut ensuite mettre à jour le schema de la base directement avec

```
Php bin/console doctrine:schema:update --dump-sql // voir les requêtes
```

```
Php bin/console doctrine:schema:update --force // exécuter les requêtes
```

Ou alors avec un système de migration à privilégier :

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

```
Php bin/console migrations:execute -u numVersion
```

e. Communiquer avec la base de données

Utilisation manuelle du manager

On peut récupérer des entités grâce au manager de doctrine. Pour une requête de sélection, on aura besoin de choisir le repository de l'entité, puis au choix d'utiliser une des 4 méthodes prédéfinies pour chaque repo :

- findAll / find
- findBy / findOneBy

Si la sélection est plus spécifique, on crée une méthode custom dans le repository de l'entité associée : on pourra mettre en place des requêtes avec des LIKE, des sous-requêtes, etc.

Utilisation des ParamConverters

Quand c'est possible, on peut aussi directement injecter les entités dans les paramètres du controller, en les mappant avec un paramètre d'url. Il suffit de typer le paramètre du controller avec la classe de l'entité. Le container de service se chargera d'appeler le manager et de renvoyer directement l'entité voulue.

f. TP

- Modéliser l'entité Article : titre, date de création, texte, en ligne, payant or not, prix
- Mettre à jour la base avec les migrations
- Créer un controller pour faire le crud de cette entité

On pourrait réaliser ce TP avec juste deux commandes, si la création automatique du crud nous convient en termes de templates/méthodes controllers :

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

```
Php bin/console make:crud
```

g. Quizz n°1

h. Lier des entités

- OneToMany / ManyToOne
- ManyToMany
- OneToOne

Voir entités User / Article

i. Cycles de vies des entités

- PrePersist / postPersist : création entité
- PreUpdate / postUpdate : mise à jour entité existante
- PreRemove / postRemove : suppression entité

Voir entité Article.php

j. Quizz 1

6. Formulaires

Cf le controller ArticleController.php, méthode new(), pour le processus de création d'une entité via un formulaire.

a. Créer un formulaire

Les formulaires dans symfony sont des classes qui doivent hériter de AstractType. On peut lier une entité à un formulaire, sans que cela ne soit obligatoire.

On peut générer un formulaire avec une commande :

php bin/console make:form

b. Liaison avec les entités

Dans la classe FormType, on va définir si le formulaire est lié ou non à une entité dans la méthode configureOptions().

c. Formulaires imbriqués

d. Personnalisation du formulaire dans le template

Dans le template twig, on peut afficher le formulaire de trois manières différentes :

- L'ensemble du formulaire avec {{ form_widget(form) }}
- Champ par champ
- Element de champ par élément de champ (label, widget, errors)

e. Validation des formulaires / gestion des erreurs

Les validations côté serveur se déclenchent en configurant les entités avec les Constraints, directement dans les annotations.

C'est le composant validator qui va rendre un formulaire / une entité invalide. Quand on utilise le composant Form, le validator est automatiquement utilisé. Mais vous pouvez très bien utiliser le validator sans formulaire pour valider une entité.

La liste des contraintes existantes :

<https://symfony.com/doc/current/reference/constraints.html>

7. Services

a. Les services

Un service dans symfony est simplement une classe. On lui délègue une tâche particulière, afin de découpler notre code (alléger le controller des logiques) et de pouvoir réutiliser ces services indépendamment.

a. Container de services

L'architecture "services" dans symfony repose sur le container de services. C'est un composant dans lequel sont chargés tous les services de l'application. Besoin d'un service ? Le container l'instancie pour nous et nous le renvoie. Si le service a déjà été instancié, la première instance est directement renvoyée.

b. Injection de dépendances

L'un des gros avantages du container est de gérer les injections de dépendances. Si un de nos services dépend d'un autre service, ce second service sera instancié automatiquement pour être injecté dans le premier.

Peu importe le nombre de dépendances de chacun de nos services, cet aspect est automatiquement géré par le container.

c. Autowiring / autoconfigure

Autowiring : Activée, cette option permet dans nos services, si nous avons besoin d'un ou plusieurs autres services, de les typer dans les paramètres du constructeur. Ils seront automatiquement injectés lors de l'instanciation du service.

Autoconfigure : Certains services ont besoin d'une configuration particulière dans les fichiers .yaml, que l'on configure avec des "tags". Cette configuration n'est pas nécessaire pour certains tags, si une classe implémente une interface particulière : cette implémentation informera le framework que d'ajouter ce tag automatiquement. Par exemple, un subscriber nécessite le tag "*kernel.event_subscriber*". L'interface *EventSubscriberInterface*, nécessaire pour un subscriber suffira à configurer ce "tag".

d. Event Dispatcher (listener / subscriber / parameters / custom Event)

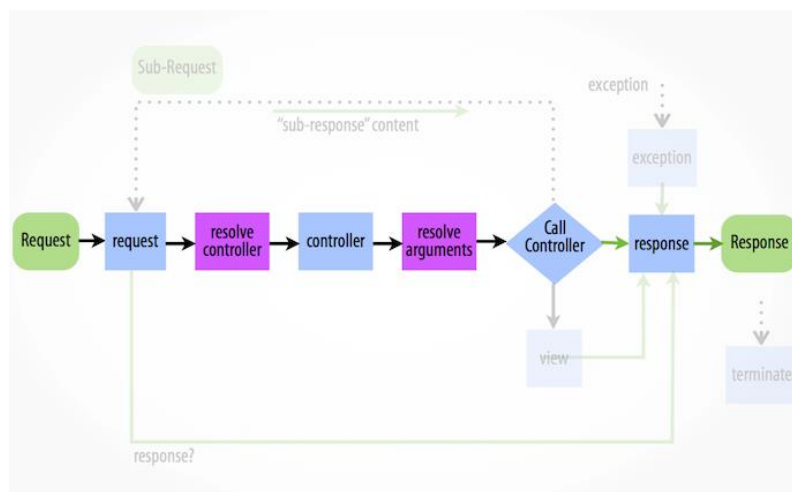
L'événement dispatcher permet de gérer des événements dans le framework. Ce répartiteur d'événements peut envoyer des événements (natifs ou métier), et ensuite des listeners/subscribers écoutant ces événements peuvent déclencher des méthodes.

On peut profiter de l'injection de dépendances dans ces listeners, comme dans tout service.

On peut passer des paramètres "textuels", si besoin aussi. Il faudra alors explicitement le service dans le fichier service.yaml.

e. Cycle de vie de symfony

Le framework suit un cycle de vie où est envoyée un événement natif à chaque étape. On peut ainsi altérer le comportement de l'application à des moments-clés du cycle.



- `kernel.request` : envoyé avant que le contrôleur ne soit déterminé.
- `kernel.controller` : envoyé après détermination du contrôleur, mais avant son exécution.
- `kernel.response` : envoyé après que le contrôleur retourne un objet `Response`.
- `kernel.terminate` : envoyé après que la réponse est envoyée à l'utilisateur.
- `kernel.exception` : envoyé si une exception est lancée par l'application.

8. Security

a. Le composant security

Le composant `security` prend en charge l'aspect authentification, et permet également de gérer les rôles des utilisateurs.

b. `Security.yml`

C'est dans ce fichier que la configuration se fera. On va créer des "firewalls", qui attrapent toutes les routes respectant un pattern, et les protège par une méthode choisie. Ainsi on peut protéger toutes les urls commençant par exemple par `"/admin"`, et autorisant uniquement certains types d'utilisateurs à se connecter sur cette espace.

Il faut bien-sûr mettre en place un formulaire dans un template.

A l'intérieur d'un firewall, il peut également y avoir des différences de rôles entre les utilisateurs : certaines pages sont accessibles avec uniquement un niveau supérieur de permissions, voire même certains éléments dans une page accessible par tous.

c. User

Les types d'utilisateurs autorisés à se connecter sur un firewall sont définis par les providers. Ces users sont des classes qui ont pour seule contrainte de devoir implémenter l'interface `UserInterface`.

d. Commande

Un système d'authentification peut entièrement être généré avec la commande :

```
php bin/console make:auth
```

e. Contrôle accès dans controllers / templates

- Controller :
 - `$this->getUser()` : récupère l'utilisateur connecté
 - `$this->isGranted("ROLE")` : teste si le user connecté possède le rôle
- Template :
 - `{{ app.user }}`
 - `{% if is_granted("ROLE") %}`

9. Custom commands

On peut créer ses propres commandes très simplement :

```
php bin/console make:command
```

Lister les commandes existantes dans le projet :

```
php bin/console list
```

10. Tests automatisés

On peut utiliser PHPUnit avec Symfony en installant la dépendance PHPUnit Bridge. Les tests automatisés consistent à tester son code. Pour le développeur il y a deux types de tests principaux :

- les tests unitaires
- les tests fonctionnels

Les tests unitaires servent à couvrir les logiques des entités et des services de façon indépendante.

Les tests fonctionnels servent à tester de façon globale l'application : dans symfony, cela correspond à tester les controllers, afin de s'assurer que les objets Response contiennent bien certains éléments html/xml attendus, le bon code http, etc.

L'environnement utilisé par les tests est un environnement à part : test. Ainsi on peut configurer les variables d'environnement dans le fichier .env.test, pour surcharger par exemple le chaîne de connexion à la base.

Par convention, on placera les tests dans le dossier *tests* à la racine du projet, et on reproduira la structure présente dans le dossier *src* : chaque classe dans le dossier *src* a sa classe test équivalente dans le dossier *tests* (dans l'idéal : il ne s'agit pas de couvrir absolument à 100% tout le code).

Cf. ArticleControllerTest.php et SlugServiceTest.php

11. Exercice final

Créer une interface de création de tickets.

Un ticket est composé :

- Titre
- Date de création / Date de dernière mise à jour
- Description
- Etat (ouvert/fermé)
- Catégorie (entité : id, name, isEnabled)
- Une liste de commentaires (entité : id, createdAt, description)

Il faut une gestion d'utilisateurs : des utilisateurs "clients" et des utilisateurs "admin". Créer simplement le système d'authentification, sans formulaire pour créer les users. Vous pourrez ajouter différents users en bdd à la mano ou via une commande.

- Les clients peuvent ouvrir un ticket via un formulaire, les admins peuvent y répondre en y ajoutant des commentaires.
- Les admins voient tous les tickets, les clients ne voit que ceux qu'ils ont créés.
- Quand un client crée un ticket ou y ajoute un commentaire, tous les admins en sont informés par mail.
- Quand un admin répond à un ticket, le client en est informé par mail.

Haut de chaque page il doit y avoir un bouton pour se connecter, et un bouton de déconnexion si on est déjà loggé. Lorsqu'une tentative de connexion échoue, il faut que le username précédemment envoyé soit pré-saisi dans le champ correspondant.