Faire de la clean architecture avec Symfony

Lorsque nombre d'entre nous avons commencé le PHP dans les années 2000, nous mélangions souvent les requêtes SQL, la récupération des paramètres de requête et l'affichage au sein des mêmes fichiers, cette pratique s'appelait le « code spaghetti ». Puis nous avons pu voir apparaître des *frameworks open source* nous proposer une manière d'améliorer l'organisation de notre code avec l'architecture MVC. Celle-ci sépare en trois couches notre code. Nous avons la partie accès aux données (*Model*), l'affichage de celles-ci en HTML (*View*) et pour coordonner le tout, une partie orchestration (*Controller*). Si sur le papier, cette architecture semble pertinente, nous avons pu rapidement en voir les limites : on a pu lire sur bon nombre de forums des développeurs demander par exemple comment appeler le code d'un *controller* à partir d'un autre, ou comment appeler le code d'un *controller* à partir d'une ligne de commande… Heureusement depuis, nous avons pu découvrir de nouveaux types d'architectures.

1. Uncle Bob

Pour ceux qui ne connaisse pas ce surnom, il désigne Robert C. Martin : ingénieur logiciel, co-auteur du manifeste Agile, mais aussi et surtout l'auteur de livres de programmation dont notamment « Clean Code » et « Clean Architecture ».

Personnellement, je l'ai découvert grâce à un collègue en 2017 via son livre « Clean Code ». Ce livre nous expose, arguments à l'appui, de bonnes pratiques à mettre en place lorsqu'on développe. Ce ne sont pas des conseils idéologiques ou théoriques, mais plutôt une forme de retour d'expérience organisé et pertinent issu de nombreuses années de développement.

On pourrait résumer son idée en une phrase : vous passerez majoritairement plus de temps à lire votre code qu'à l'écrire, donc investissez dans l'écriture pour simplifier la future lecture pour vos collègues et vous-même quand vous repasserez sur votre projet plusieurs mois/années après sa mise en production.

Investissez du temps pour rendre votre code le plus lisible possible, vous vous remercierez plus tard.

Je listerai quelques conseils que je vous invite à adopter :

- évitez au maximum de commenter, préférez des noms de variables et méthodes explicites : si vous, ou un collègue intervenez sur du code, il ne pensera pas forcément à modifier le/les commentaires. Il est mieux d'avoir un nom de variable/fonction/méthode compréhensible, plutôt qu'un nom court avec un commentaire indiquant ce qu'elle contient/fait ;
- ne codez pas des méthodes/fonctions gigantesques qui font tout, découpez et limitez un maximum le périmètre de chacune : moins de code, plus de lisibilité, moins de risque d'erreur ;
- évitez les nombres magiques comme `if($user->age > 18)`, mais plutôt une méthode parlante `if($this->isUserLegalAge($user) )` et utilisez des constantes que vous écrirez tout en haut de vos classes plus facilement visibles ;
- essayez autant que possible de couvrir votre code métier par des tests unitaires, cela vous permettra de sécuriser vos refactorisations futures.

2. Clean, Hexagonal, Onion Architecture… même combat

Après nous avoir proposé un livre pour nous aider à améliorer la lisibilité de notre code, le même auteur a écrit quelques années après un livre pour faire de même avec l'architecture globale de nos applications.

Son idée s'appuie sur les travaux précédents : « Hexagonale » de A. Cockburn, « Onion » de J. Pallermo… qui au fond partagent la même idée : séparer le code métier de son implémentation technique.

En effet, contrairement à ce que l'on ferait naturellement avec un *framework* de type MVC comme Symfony et consort, on ne souhaite pas diviser entre données/présentation/orchestrateur.

L'idée ici est plutôt d'avoir le code métier d'une part, et l'implémentation d'autre part avec comme liaison entre ces deux mondes des ports et des adaptateurs.

On peut voir sur l'image 1 la *clean architecture* d'*Uncle Bob*, et sur l'image 2 l'architecture hexagonale.

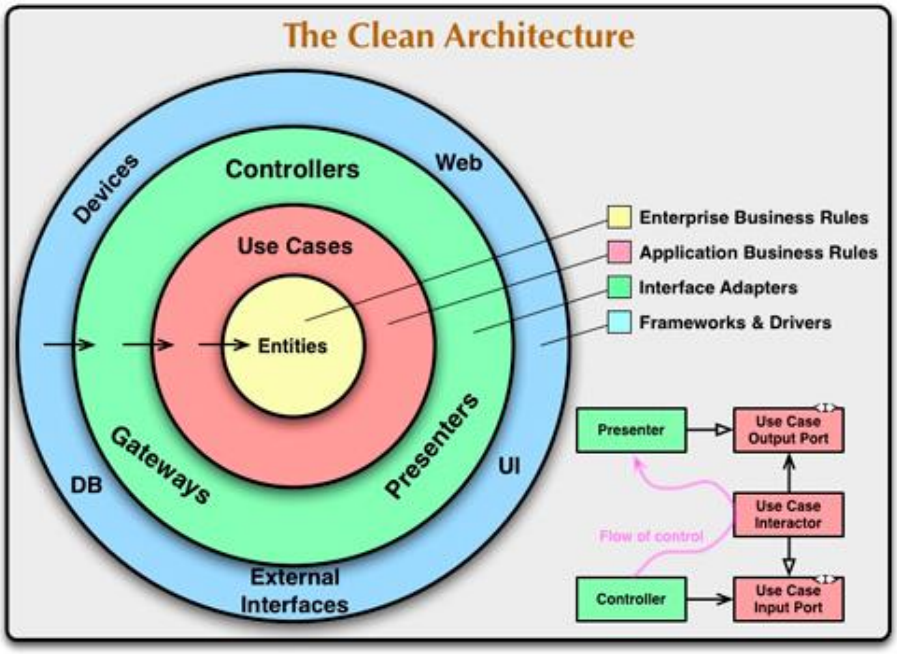

Fig. 1 : La clean architecture selon Uncle Bob.

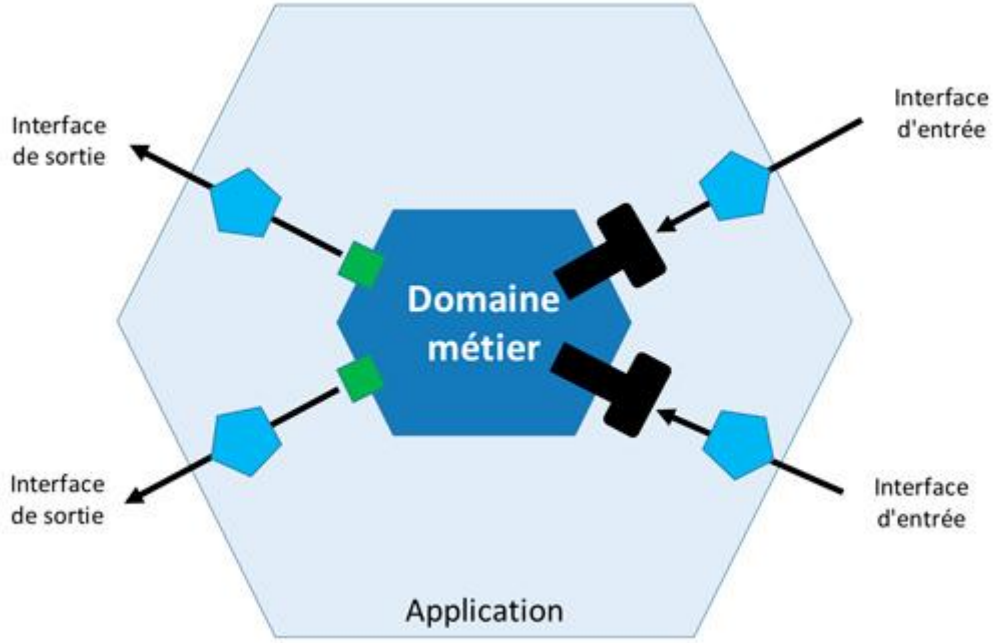

Fig. 2 : L'architecture hexagonale.

Ce type d'architecture permet :

- d'avoir un code indépendant du *framework*/librairies et de ses versions ;
- à l'ensemble du code métier d'être facilement testable (sans besoin de base de données, d'API...) ;
- d'être indépendant de la présentation, car on peut facilement ajouter de nouvelles entrées et de nouvelles sorties (Web, API, ligne de commande...) ;

- d'être indépendant du système de stockage de données : on peut partir d'un fichier plat pour évoluer vers une base de données SGBD ou NoSQL ou encore une API.

Clean architecture et Symfony compatible ?

Si le *framework* propose depuis des années une approche orientée MVC, on peut assez facilement changer le paramétrage par défaut pour obtenir une architecture *clean*/hexagonale.

3. Installation

Pour installer Symfony, je vous invite soit à ajouter le dépôt Symfony à votre gestionnaire de sources, puis installer leur binaire de ligne de commande via les commandes suivantes :

```
echo 'deb [trusted=yes] https://repo.symfony.com/apt/ /' | sudo tee
/etc/apt/sources.list.d/symfony-cli.list
sudo apt update
sudo apt install symfony-cli
```

Ou soit à vous rendre à l'adresse : https://github.com/symfony-cli/symfony-cli/releases

Vous y trouverez des paquets d'installation **`.deb`**, **`.rpm`**... ou bien des archives au format **`.tag.gz`**, téléchargez puis désarchivez, par exemple en ligne de commande :

```
cd /tmp
wget                          https://github.com/symfony-cli/symfony-
cli/releases/download/v5.1.0/symfony-cli_linux_386.tar.gz
tar -xvf symfony-cli_linux_386.tar.gz
mv symfony ~/bin
```

Note : par habitude, je copie tous mes exécutables dans le répertoire **`bin`** de mon répertoire **`home`**.

4. Création de l'application

Créons une application vierge :

```
mkdir -p /home/mika/www/LM/CleanArchiSf
cd /home/mika/www/LM/CleanArchiSf
symfony new myProject
cd myProject
```

Après cette commande de création, nous avons juste le squelette de notre application avec des répertoires **src**, **config**, **public**, et notre répertoire contenant les dépendances *composer* dans *vendor*.

5. Réorganisation clean architecture

5.1 Création de la nouvelle arborescence principale

Nous souhaitons diviser en deux parties notre application : la partie métier que nous appellerons **domain** et la partie extérieure/concrète liée aux bases de données, système de fichiers... que nous nommerons **infrastructure**.

Pour cela, nous devons créer les arborescences avec les commandes suivantes :

```
mkdir src/Domain
mkdir src/Infrastructure
mkdir src/Infrastructure/Symfony
mv src/Controller src/Infrastructure/Symfony
```

Nous commençons par créer dans infrastructure un sous-répertoire Symfony qui contiendra toute la partie propre au *framework* et nous y déplaçons notre répertoire **controller**.

On peut aussi sortir totalement notre code métier dans une autre arborescence que celle de notre projet Symfony et avoir ainsi deux projets complètement indépendants. Cela permettrait par la suite de travailler à une autre implémentation dans un second projet simplement en incluant notre projet **domain** (par exemple lors d'une montée en version du *framework*).

5.2 Adaptation de l'autoloader

Ceci fait, il nous faut indiquer dans notre fichier **composer** le changement de chemins de nos éléments :

- Fichier **composer.json** :
  - Remplacez :

```
"autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    },
```

  -
  - Par :

```
"autoload": {
        "psr-4": {
            "Domain\\": "src/Domain/",
            "Infrastructure\\": "src/Infrastructure/"
        }
    },
```

-
  - Puis :

```
"autoload-dev": {
        "psr-4": {
            "App\\Tests\\": "tests/"
        }
    },
```

-
  - À remplacer par :

```
    "autoload-dev": {
        "psr-4": {
            "Tests\\": "tests/"
        }
    },
```

5.3 Modification des chemins dans les fichiers de configuration

Modifions ensuite les chemins des services :

- Fichier **`config/services.yaml`** :
  - Remplacez :

```
    App\:
        resource: '../src/'
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'
            - '../src/Tests/'
```

-
  - Par :

```
    Domain\:
        resource: '../src/Domain'
        exclude:
            - '../src/Domain/Entity/'

    Infrastructure\:
        resource: '../src/Infrastructure/'
        exclude:
            - '../src/Infrastructure/Symfony/Kernel.php'

    Infrastructure\Symfony\Controller\:
        resource: '../src/Infrastructure/Symfony/Controller/'
```

```
        tags: ['controller.service_arguments']
```

5.4 Déplacement du Kernel

Déplaçons le fichier **`Kernel.php`** dans l'infrastructure :

```
mv src/Kernel.php src/Infrastructure/Symfony
```

Une fois déplacé, il va falloir intervenir sur **`bin/console`** et **`public/index.php`** qui en dépendent.

5.5 Modification des namespaces d'infrastructure

Modifiez juste le **`namespace`** utilisé pour charger le *Kernel* d'**`App\`** vers **`Infrastructure\Symfony\`** :

- Fichier **`bin/console`** :
  - Remplacez :

```
use App\Kernel;
```

  -
  - Par :

```
use Infrastructure\Symfony\Kernel;
```

Idem dans le fichier dans public :

- Fichier **`public/index.php`** :
  - Remplacez :

```
use App\Kernel;
```

  -
  - Par :

```
use Infrastructure\Symfony\Kernel;
```

Et la classe **`Kernel`** elle-même :

- Fichier **`src/Infrastructure/Symfony/Kernel.php`** :
  - Remplacez :

```
namespace App;
```

  -
  - Par :

```
namespace Infrastructure\Symfony;
```

6. Passons à la pratique

6.1 Ajoutons quelques librairies avant de commencer

Cette étape permet de voir comment réagir une fois notre architecture installée à l'ajout de nouveaux composants d'infrastructure. En effet, tout ce qui sera installé via **composer** devra être déplacé, et sa configuration adaptée pour fonctionner correctement avec cette nouvelle architecture.

Pour la partie présentation, nous utiliserons dans un premier temps le moteur de *template* **Twig**.

Installez-le sur votre projet via la commande **composer** :

```
symfony composer require symfony/twig-bundle
```

Par défaut, **composer** va installer le répertoire des *templates* à la racine du projet, avant le répertoire **src**, or pour nous, cela concerne l'infrastructure, nous devons donc le déplacer dans **src/Infrastructure/Twig** :

```
mkdir -p src/Infrastructure/Twig
mv templates src/Infrastructure/Twig
```

Changez le fichier de config :

- Fichier **config/packages/twig.yaml** :
    - Remplacez :

```
twig:
    default_path: '%kernel.project_dir%/templates'
```

    -
    - Par :

```
twig:
    default_path: '%kernel.project_dir%/src/Infrastructure/Twig/templates'
```

Nous pouvons également installer le paquet **form** qui nous servira pour le traitement des formulaires via la commande :

```
symfony composer require symfony/form
```

Pas besoin pour ce composant d'intervenir dans le fichier de configuration.

J'ai commencé ici par installer ces librairies, mais à part le paramétrage dans l'*autoloader* du chemin pour trouver notre partie domaine, toute la partie infrastructure peut être faite bien après.Mais en procédant ainsi, cela peut rassurer d'avoir déjà nos briques d'implémentation déjà initiées pour se dire « OK, j'ai mon code métier ici, et mes vues, mes requêtes... de ce côté ».

Je fais ici exprès de ne pas installer la partie base de données, nous verrons la facilité justement de passer du modèle de stockage de données au départ du projet, au définitif.

6.2 Présentation du projet à développer

Nous allons développer une petite boutique en ligne, nous pourrons ajouter des produits à la vente d'un côté, et les ajouter dans un panier afin de passer commande de l'autre. Nous ne verrons pas ici l'ensemble du code de l'application, mais quelques parties pour présenter cette architecture. Vous trouverez le reste du code source sur le dépôt GitHub de l'article.

6.3 L'organisation par use Case

Imaginez si vous deviez faire cette application en entreprise, vous auriez une équipe projet, peut-être une maîtrise d'ouvrage ou un « *product owner* ». Avec lui, vous auriez découpé ce projet en plusieurs parties, et à l'intérieur de chaque regroupement, vous auriez créé des « *User stories* » avec éventuellement des taches/sous-tâches associées.

Avec ce type d'architecture, comme en DDD (*Domain Driven Development*), nous allons le plus possible tant par le vocabulaire que la façon d'écrire notre code coller au maximum au métier.

Pour rappel, en *clean architecture*, on sépare le code métier de l'infrastructure. Nous allons commencer ici par cette partie métier, sans nous soucier de la partie implémentation : pas besoin à ce moment de savoir si nous stockerons les données dans une base de données ou dans des fichiers JSON, ou via une API. Idem pour l'affichage des pages via un moteur de *template* type Twig, ou via une API REST / GraphQL avec une interface *frontend* développée en ReactJs / Vue.js...

Notre premier cas d'usage, ou « *use case* » consistera à permettre à notre service produit d'enregistrer un nouveau produit avec ses caractéristiques.

Dans notre architecture, nous aurons besoin de créer plusieurs choses :

- des entités au sens métier (et non ORM) ;
- des cas d'usage ;
- des requêtes envoyées à nos cas d'usage ;
- des réponses retournées par nos cas d'usage ;
- enfin des présentateurs pour mettre en forme la réponse.

Pour l'organisation, vous pouvez faire une arborescence par entité générale, et avoir dans chacune l'ensemble de ces types, ou vous pouvez ranger vos classes par type.

Ou encore, regrouper dans des sous-répertoires du nom de votre cas d'usage regroupé sa requête, sa réponse, sa présentation et le cas d'usage lui-même... comme cela vous arrange.

Ici, l'arborescence qui va être utilisée est un exemple, à vous de vous organiser aussi confortablement que possible. Ce type d'architecture oblige déjà à une certaine contrainte d'organisation, ne soyez pas dogmatique dans votre approche, placez le curseur au bon endroit pour bénéficier des avantages, en limitant les contraintes au quotidien.

6.4 Commençons par un premier cas d'usage

Partons sur l'enregistrement de produits dans le catalogue du magasin.

Nous pouvons créer dans notre répertoire `Domain` l'arborescence suivante :

- `Entity` : pour nos entités métier (pas au sens ORM/base, mais bien métier) ;
- `Gateway` : pour nos contrats d'interface permettant de récupérer/interagir avec la donnée, le serveur de fichier, les mails… ;
- `UseCase` : nos fameux cas d'usage, la majeure partie de nos développements ;
- `Request` : l'objet reçu par nos classes `UseCase` pour définir les paramètres d'entrée, pas `request` au sens HTTP, mais plutôt requête au sens paramètres de l'action ;
- `Response` : l'objet qui sera renvoyé par notre `UseCase` qui contiendra à la fois le statut, mais également les divers éléments retournés ;
- `Presenter` : une interface cette fois qui devra être implémentée côté infrastructure pour faire le rendu en fonction du type d'application (Web, API REST/Soap/GraphQL…).

Nous ne mettrons pas à la racine du répertoire cas d'usage notre premier, sinon à la fin du projet nous aurons une quinzaine/vingtaine de *use cases* situés au même endroit, sans aucune logique.

Nous allons ici initier plusieurs regroupements pour cette boutique, libre à vous de choisir les regroupements qui vous correspondent le plus.

Nous aurons ici :

- *Sale* : toutes les actions liées à la vente de produits ;
- *Buy* : la partie permettant aux clients d'acheter les produits ;
- *Customer-area* : qui permettra aux clients le suivi de leurs commandes ;
- *Shipping* : qui permettra à l'équipe logistique d'envoyer les colis des commandes.

Nous pouvons déjà créer l'arborescence pour ce premier regroupement qui accueillera notre premier développement :

```
mkdir -p src/Domain/Entity/
mkdir -p src/Domain/UseCase/Sale/
mkdir -p src/Domain/Request/Sale/
mkdir -p src/Domain/Response/Sale/
mkdir -p src/Domain/Presenter/Sale/
```

Regardons notre premier cas d'usage :

- Fichier **src/Domain/UseCase/Sale/AddProductToSellUseCase.php** :

```
<?php

namespace Domain\UseCase\Sale;

use Domain\Entity\Product;
use Domain\Entity\VATRate;
use Domain\Gateway\CategoryGatewayInterface;
use Domain\Gateway\ProductGatewayInterface;
use Domain\Gateway\VATRateGatewayInterface;
use Domain\Presenter\Sale\AddProductToSellPresenterInterface;
use Domain\Request\Sale\AddProductToSellRequest;
use Domain\Response\Sale\AddProductToSellResponse;
use Domain\Tool\Validator;
use Domain\Tool\ValidatorResponse;

class AddProductToSellUseCase{

   protected ProductGatewayInterface $productGateway;
   protected VATRateGatewayInterface $vatRateGateway;
   protected CategoryGatewayInterface $categoryGateway;

   protected Validator $validator;

   public function __construct(
          ProductGatewayInterface $productGateway,
          VATRateGatewayInterface $vatRateGateway,
          CategoryGatewayInterface $categoryGateway,
          Validator $validator
   ){
     $this->productGateway=$productGateway;
     $this->vatRateGateway=$vatRateGateway;
     $this->categoryGateway=$categoryGateway;

     $this->validator=$validator;

   }

   public function execute(AddProductToSellRequest $request) : AddProductTo
SellResponse{

   try{
    $response=new AddProductToSellResponse();

     $productRequestValidation=$this->getProductRequestValidation($this->val
idator,$request);
       if(!$productRequestValidation->isValid()){
```

```
            $response->setStatusNotValid();

            foreach($productRequestValidation->getErrorList() as $field => $
fieldErrorList){
                    $response->addFieldErrorMessage($field,join(',',$fieldEr
rorList));
            }
            return $response;

        }

        if($this->dontFindCategoryWithThisId($request->getProductToAdd()->ca
tegory_id) ){
        $response->addFieldErrorMessage('category_id','Unable to find Catego
ry with id '.$request->getProductToAdd()->category_id);

        $response->setStatusNotValid();

        return $response;
        }

        $vatRate=$this->findVatRateById($request->getProductToAdd()->vatRate
_id);
        if($vatRate===null){
        $response->addFieldErrorMessage('vatRate_id','Unable to find VAT Rat
e with id '.$request->getProductToAdd()->vatRate_id);

        $response->setStatusNotValid();

        return $response;
        }

         $productToAdd=$this->generateProductToAdd($request->getProductToAdd(
),$vatRate);

        $productAddResponse=$this->productGateway->add($productToAdd);
        if(!$productAddResponse->isStatusSuccess()){

            $response->setStatusFailed();
            $response->setFailedMessage('Error when ask to add product with G
ateway:'
        .$productAddResponse->getFailedMessage());

         return $response;
         }

         $response->setStatusSuccess();
         $response->setSuccessMessage('Product added with success');

         return $response;


    }catch(\Exception $e){

         $response=new AddProductToSellResponse();
         $response->setStatusException();
         $response->setException($e);

         return $response;
     }
```

```
  }

  public function generateProductToAdd(Product $productToAdd,$vatRate){

     $productToAdd->calculateIncTaxSellingPrice($vatRate->rate);
     $productToAdd->hide();

     return $productToAdd;
  }

  public function dontFindCategoryWithThisId($categoryId){

     $rawCategory=$this->categoryGateway->findById($categoryId);
     if($rawCategory===null){
         return true;
     }
     return false;
  }

  public function findVatRateById($vatRageId){
      $rawVatRate=$this->vatRateGateway->findById($vatRageId);
      if($rawVatRate===null){
          return null;
      }
      return new VatRate((array)$rawVatRate);
  }


  public function getProductRequestValidation(Validator $validator,AddProdu
ctToSellRequest $request): ValidatorResponse{

     $productToAdd=$request->getProductToAdd();

     if($productToAdd==null){
        $validator->addErrorOnField('global','You must provide a product');
        return $validator->validate();
     }

     $validator->load($productToAdd);

     $validator->assertIsNotEqual('category_id',0,'should_select_one');
     $validator->assertIsNotEqual('vatRate_id',0,'should_select_one');

     $validator->assertIsNotEmpty('title','should_fill_it');

     $validator->assertIsNotEmpty('excTaxBuyingPrice','should_fill_it');
     $validator->assertIsFloat('excTaxBuyingPrice','foat needed');

     $validator->assertIsNotEmpty('excTaxSellingPrice','should_fill_it');
     $validator->assertIsFloat('excTaxSellingPrice','foat needed');

     $validator->assertIsNotEmpty('description','should_fill_it');
     $validator->assertIsNotEmpty('stock','should_fill_it');
     $validator->assertIsInteger('stock','should_be_integer');


     return $validator->validate();
  }
}
```

Ceci est uniquement le code demandé par le métier, il contient principalement deux méthodes : un constructeur qui permet de récupérer les dépendances extérieures, les classes « outils » nécessaires à cette action et une méthode **`execute`** qui permettra comme son nom l'indique d'effectuer l'action demandée avec en paramètre l'objet **`request`** contenant les paramètres d'entrée, elle nous retournera un objet **`response`** contenant à la fois le statut retour et divers éléments pertinents en fonction de l'action demandée.

```
public function __construct(
                ProductGatewayInterface $productGateway,
                VATRateGatewayInterface $vatRateGateway,
                CategoryGatewayInterface $categoryGateway,
                Validator $validator
    ){
```

Faisons un petit aparté sur ce constructeur : comme vous pouvez le voir, nous recevons ici nos objets « *gateway* » qui nous permettent d'interagir avec nos données, mais on précise ici que l'on attend des interfaces et non des classes.

Avec cette architecture, nous n'attendons pas des objets spécifiques, mais des objets respectant les contrats, ceci nous permet d'être indépendants de l'implémentation : nous pouvons très bien commencer par une implémentation temporaire retournant de simples fichiers JSON pour une démonstration et attendre la mise en place de la base de données. Puis, par la suite décider de passer par une API. Tous ces changements n'affectant pas votre code métier, juste la partie infrastructure.

```
public function execute(AddProductToSellRequest $request) : AddProductToSel
lResponse{

  try{
     $response=new AddProductToSellResponse();

     $productRequestValidation=$this->getProductRequestValidation($this->va
lidator,$request);
     if(!$productRequestValidation->isValid()){

     $response->setStatusNotValid();

     foreach($productRequestValidation->getErrorList() as $field => $fieldE
rrorList){
     $response->addFieldErrorMessage($field,join(',',$fieldErrorList));
     }
     return $response;

     }

      if($this->dontFindCategoryWithThisId($request->getProductToAdd()->cate
gory_id) ){
     $response->addFieldErrorMessage('category_id','Unable to find Category
with id '.$request->getProductToAdd()->category_id);

     $response->setStatusNotValid();

     return $response;
     }
```

```
      $vatRate=$this->findVatRateById($request->getProductToAdd()->vatRate_i
d);
     if($vatRate===null){
     $response->addFieldErrorMessage('vatRate_id','Unable to find VAT Rate w
ith id '.$request->getProductToAdd()->vatRate_id);

     $response->setStatusNotValid();

     return $response;
     }

    ...
```

Notre méthode d'exécution, quant à elle, reçoit une requête comprenant l'objet produit que nous souhaitons ajouter.

Après avoir validé les valeurs envoyées, nous vérifions que les identifiants externes comme celui du taux de TVA ou celui de la catégorie existent bien, on retournera une réponse d'erreur dans le cas contraire.

```
$productToAdd=$this->generateProductToAdd($request->getProductToAdd(),$vatR
ate);
```

Si l'ensemble est correct, on peut générer l'objet produit à ajouter. Ici, nous ne faisons que calculer le prix TTC et forcer son statut caché pour éviter de le faire apparaître trop tôt sur le catalogue. Mais notez que l'on passe par une méthode pour le faire, ceci facilitera les tests unitaires. En effet, on applique au mieux les principes du *clean code*, à savoir de limiter au maximum le nombre de lignes de chaque méthode facilitant leur test, et limitant les éventuels *bugs* : il est clair que plus une méthode contient de code, plus on risque d'y faire des erreurs.

```
public function generateProductToAdd(Product $productToAdd,$vatRate){

        $productToAdd->calculateIncTaxSellingPrice($vatRate->rate);
        $productToAdd->hide();
        return $productToAdd;
    }
```

Dans celle-ci, on voit que l'on demande à l'entité de calculer le prix TTC à l'aide du taux, puisqu'on lui demande de se cacher.

On pourrait directement le calculer dans cette méthode, mais on risquerait de créer un changement de comportement entre un cas d'ajout de produit et celui de modification de celui-ci, on décide donc de centraliser cette modification dans l'entité, on aura ainsi le même code utilisé à chaque fois.

Regardons justement cette entité :

- Fichier **`Domain/Entity/Product.php`** :

```
<?php

namespace Domain\Entity;
```

```
class Product extends AbstractEntity{

    public $id=null;
    public $title=null;
    public $description=null;

    public $excTaxBuyingPrice=null;
    public $excTaxSellingPrice=null;

    public $incTaxSellingPrice=null;

    public $stock=null;

    public $visible=null;

    public $category_id=null;
    public $vatRate_id=null;

    public function calculateIncTaxSellingPrice($vatRate){
        $this->incTaxSellingPrice=($this->excTaxSellingPrice*(1+$vatRate));
    }
    public function hide(){
        $this->visible=false;
    }
    public function display(){
        $this->visible=true;
    }

}
```

Cette classe contient les propriétés pertinentes à manipuler, ainsi que certaines méthodes pour interagir.

Regardons également la classe requête qui sera envoyée :

- Fichier **`src/Domain/Request/Sale/AddProductToSellRequest.php`** :

```
<?php

namespace Domain\Request\Sale;

use Domain\Entity\Product;

class AddProductToSellRequest{

    protected ?Product $product=null;

    public function setProductToAdd(Product $product){
        $this->product=$product;
    }
    public function getProductToAdd():?Product{
        return $this->product;
    }

}
```

Dans notre cas, nous avons uniquement besoin des informations d’un produit à ajouter.

Concernant notre classe en général, il est intéressant ici de ne pas retourner juste un booléen de bonne exécution, mais de pouvoir retourner une véritable réponse complexe avec un statut en fonction des différents cas qui peuvent arriver, des données fournies non correctes/exhaustives, au champ externe non trouvé en retournant proprement une réponse identifiée.

Ensuite, en fonction de l'implémentation (Web, API, ligne de commande…), la partie infrastructure se chargera du traitement de ces erreurs.

Notez également que pour notre partie persistance de la donnée, on ne s'attend pas forcément à une réponse positive de la **gateway**, on gère le retour de demande de données et ainsi on gère proprement les éventuels soucis pendant le déroulement de notre **useCase**, plutôt que de se retrouver au moindre souci à une exception SQL, API… qu'il faudra interpréter, ici on peut retourner à chaque étape une réponse précise sur le problème rencontré qui pourrait permettre de faciliter le support de production dans une autre partie (émettre un appel à un service d'*alerting*, d'envoi de *mail* au support...).

Enfin, nous retournons ici une réponse, car c'est un **useCase** d'action qui ne devrait rien avoir à afficher, ici la réponse nous permet juste de nous indiquer si elle s'est bien passée, si on peut rediriger sur la page suivante...

En revanche, pour un **useCase** plutôt orienté affichage, comme celui du panier, ou d'une fiche produit, dans ce cas, on aura une légère différence, par exemple :

- Fichier **src/Domain/UseCase/Sale/ListProductToSellUseCase.php** :

```
<?php

namespace Domain\UseCase\Sale;

use Domain\Gateway\CategoryGatewayInterface;
...
use Domain\Presenter\Sale\ListProductToSellPresenterInterface;
...

class ListProductToSellUseCase{

...

    public function execute(ListProductToSellRequest $request, ListProductT
oSellPresenterInterface $presenter) {

        $rawProductList=$this->productGateway->findAll();

        $productList=[];
        foreach($rawProductList as $productLoop){
            $productList[]=(object)[
                'id'=>$productLoop->id,
                'title'=>$productLoop->title,
                'excTaxBuyingPrice'=>  $this->formatPrice($productLoop->excT
axBuyingPrice),
```

```
                'excTaxSellingPrice'=>  $this->formatPrice($productLoop->exc
TaxSellingPrice),
                'incTaxSellingPrice'=>  $this->formatPrice($productLoop->inc
TaxSellingPrice),
                'stock'=> $productLoop->stock,

            ];

        }

        $response=new ListProductToSellResponse();
        $response->setStatusSuccess();
        $response->setProductList($productList);

        return $presenter->present($response);
    }
```

On voit ici que l'on passe une interface de présentation en second argument d'exécution. En effet, lors de l'implémentation, on enverra au **useCase** en fonction du besoin une classe de présentation JSON, HTML...

Pas besoin ici d'intercepter la réponse et d'aviser, on se contente de passer la responsabilité à la couche de présentation de se charger du rendu.

6.5 Après le code, le test

Notre code écrit, nous allons ajouter un test unitaire pour valider ce premier cas et sécuriser ses futures évolutions.

Commençons par créer l'arborescence qui accueillera nos tests unitaires de ce premier regroupement **Sale** :

```
mkdir -p Tests/unitTests/Sale
```

Et voici notre premier test unitaire :

- Fichier **Tests/unitTests/Sale/AddProductToSellUseCaseTest.php** :

```
<?php

declare(strict_types=1);

use Domain\Entity\Category;
use Domain\Entity\Product;
use Domain\Entity\VATRate;
use Domain\Gateway\ProductGatewayInterface;
use Domain\Gateway\VATRateGatewayInterface;
use Domain\Gateway\CategoryGatewayInterface;
use Domain\Gateway\GatewayResponse;
use Domain\Request\Sale\AddProductToSellRequest;
use Domain\Response\Sale\AddProductToSellResponse;
use Domain\Tool\Validator;
use Domain\UseCase\Sale\AddProductToSellUseCase;
use PHPUnit\Framework\TestCase;
```

```
final class AddProductToSellUseCaseTest extends TestCase{

  public function test_executeShouldFinishSuccess(){

    $productGateway=$this->createMock(ProductGatewayInterface::class);
    $vatRateGateway=$this->createMock(VATRateGatewayInterface::class);
    $categoryGateway=$this->createMock(CategoryGatewayInterface::class);

    $categoryFound=new Category(['id'=>101,'name'=>'Category 1']);
    $categoryGateway->method('findById')->willReturn($categoryFound);

    $vatRateFound = new VATRate(['id'=>1001,'rate'=>0.196]);
    $vatRateGateway->method('findById')->willReturn($vatRateFound);

    $validator=new Validator();

    $createNewProductUseCase=new  AddProductToSellUseCase($productGateway,$v
atRateGateway,$categoryGateway,$validator);

    $request=new AddProductToSellRequest();
    $request->setProductToAdd(new Product([
        'title'=>'Clef usb',
        'excTaxBuyingPrice'=>4.5,
        'excTaxSellingPrice'=>8.0,
        'stock'=>20,
        'description'=>'Clef usb 8GO',
        'category_id'=>1,
        'vatRate_id'=>1
    ]));

    $productAddResponse=new GatewayResponse();
    $productAddResponse->setStatusSuccess();
    $productGateway->method('add')->willReturn($productAddResponse);

    $response=$createNewProductUseCase->execute($request);

    $expectedResponse=new AddProductToSellResponse();
    $expectedResponse->setStatusSuccess();
    $expectedResponse->setSuccessMessage('Product added with success');

    $this->assertEquals($expectedResponse,$response );
 }

 public function test_generateProductToAddShouldFinishSuccess(){

    $productGateway=$this->createMock(ProductGatewayInterface::class);
    $vatRateGateway=$this->createMock(VATRateGatewayInterface::class);
    $categoryGateway=$this->createMock(CategoryGatewayInterface::class);

    $validator=new Validator();

     $createNewProductUseCase=new  AddProductToSellUseCase($productGateway,$v
atRateGateway,$categoryGateway,$validator);

    $requestProduct=new Product([
        'title'=>'Clef usb',
        'excTaxBuyingPrice'=>4.5,
        'excTaxSellingPrice'=>8.0,
        'stock'=>20,
        'description'=>'Clef usb 8GO',
        'category_id'=>1,
```

```
        'vatRate_id'=>50
     ]);

    $vatRate = new VATRate(['id'=>1001,'rate'=>0.196]);

    $productToAdd=$createNewProductUseCase->generateProductToAdd($requestPr
oduct,$vatRate);

    $incTaxSellingPriceCalculated=9.568;//0.8 x (1+0.196)

    $expectedProductToAdd=new Product([
        'title'=>'Clef usb',
        'excTaxBuyingPrice'=>4.5,
        'excTaxSellingPrice'=>8.0,
        'stock'=>20,
        'description'=>'Clef usb 8GO',
        'category_id'=>1,
        'vatRate_id'=>50,

        'incTaxSellingPrice'=>$incTaxSellingPriceCalculated,
        'visible'=>false
    ]);

    $this->assertEquals($expectedProductToAdd,$productToAdd );
  }

}
```

Ici, comme en architecture hexagonale, le fait de fournir les éléments d'infrastructure à notre code métier permet de simuler tous les cas désirés, et ceci très facilement : juste besoin de créer des *mocks* avec uniquement les interfaces. Rappelons qu'à ce stade, aucune base de données n'a encore été choisie, et encore moins installée.

Vous noterez que l'on peut dans le deuxième test très facilement tester notre création de produits avec la méthode dédiée, sans avoir besoin ici de créer des *mocks*.

7. L'infrastructure

Nous souhaiterions utiliser ici les annotations.

Ajoutons la dépendance nécessaire via la commande :

```
symfony composer require annotations
```

Il faut indiquer où trouver nos annotations dans notre nouvelle arborescence :

- Fichier **`config/routes/annotations.yaml`** :

```
controllers:
    resource: ../../src/Infrastructure/Symfony/Controller/
    type: annotation
```

Modifiez / vérifiez la valeur du fichier :

- Fichier **config/packages/sensio_framework_extra.yaml** :

```
sensio_framework_extra:
    router:
        annotations: true
```

Ceci paramétré, nous allons pouvoir nous attaquer à l'implémentation.

7.1 API REST

À partir de là, vous pouvez créer une première route REST utilisant ce cas d'usage :

Créer l'arborescence :

```
mkdir -p Infrastructure/Symfony/Controller/API
```

- Fichier **src/Infrastructure/Symfony/Controller/API/ProductController.php** :

```
<?php

namespace Infrastructure\Symfony\Controller\API;

use Domain\Entity\Product;
use Domain\Gateway\CategoryGatewayInterface;
use Domain\Gateway\ProductGatewayInterface;
use Domain\Gateway\VATRateGatewayInterface;
use Domain\Request\Sale\AddProductToSellRequest;
use Domain\Tool\Validator;
use Domain\UseCase\Sale\AddProductToSellUseCase;
use Infrastructure\Symfony\Presenter\Sale\AddProductToSellPresenterToJson;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;


class ProductController extends AbstractController{


 /**
 * @Route("/API/product/add",methods={"POST"})
 */
 public function addAction(
    Request $request,
    ProductGatewayInterface $productGateway,
    VATRateGatewayInterface $vatGateway,
    CategoryGatewayInterface $categoryGateway,
    AddProductToSellPresenterToJson $AddProductToSellPresenterToJson

    ){

   $jsonRequest=json_decode($request->getContent());

   $useCaseRequest=new AddProductToSellRequest();
```

```
    $useCaseRequest->setProductToAdd(new Product((array)$jsonRequest->produc
t));

    $useCase=new AddProductToSellUseCase($productGateway,$vatGateway,$catego
ryGateway,new Validator());
    $response=$useCase->execute($useCaseRequest);


    return $AddProductToSellPresenterToJson->present($response);

   }

}
```

Pour notre première route infrastructure, nous nous retrouvons dans l'univers de **Symfony**, cette fois utilisons l'ensemble d'outils que nous propose ce *framework* pour proposer une API REST. Nous réceptionnerons en méthode POST un contenu JSON et nous retournerons la réponse au même format.

Tout d'abord, nous récupérons, puis décodons le JSON, nous constituons ensuite l'objet **Request** demandé par notre **UseCase**, et une fois exécuté, nous envoyons la **Response** à un **Presenter** JSON qui se chargera de le restituer.

Créons l'arborescence de nos classes de présentation :

```
mkdir Infrastructure/Symfony/Presenter/Sale
```

Et notre première classe pour présenter en JSON :

- Fichier **src/Infrastructure/Symfony/Presenter/Sale/AddProductToSellPresenterToJson.php** :

```
<?php

namespace Infrastructure\Symfony\Presenter\Sale;

use Domain\Presenter\Sale\AddProductToSellPresenterInterface;
use Domain\Response\Sale\AddProductToSellResponse;
use Symfony\Component\HttpFoundation\JsonResponse;

class AddProductToSellPresenterToJson implements AddProductToSellPresenterI
nterface{

  public function present(AddProductToSellResponse $response)
  {

    $statusCode=500;
    $content=(object)[];

    if($response->isStatusSuccess()){
       $statusCode=200;
       $content->message = $response->getSuccessMessage();

    }else if($response->isStatusFailed() ){
       $statusCode=500;
       $content->error=$response->getFailedMessage();
    }else if($response->isStatusException()){
```

```
        $statusCode=500;
        $content->error=$response->getException()->getMessage();

        //execute needed action with exception or failed message (log, mail t
o support...)
    }else if($response->isStatusNotValid()){
        $statusCode=400;

        $content->message= 'You have some errors in your request';
        $content->errors=$response->getFieldErrorMessageList();
    }else{

        $statusCode=500;
        $content->message='unknown status';
    }

    $jsonResponse=new JsonResponse();
    $jsonResponse->setStatusCode($statusCode);
    $jsonResponse->setContent( json_encode($content));

    return $jsonResponse;
  }

}
```

On peut tester cette première implémentation en envoyant avec **insomnia**, **postman** ou **curl** le JSON suivant :

```
curl --request POST \
  --url http://clean-archi.local/API/product/add \
  --header 'Content-Type: application/json' \
  --data '{
    "product":{
            "title":"test",
            "category_id":1,
            "excTaxBuyingPrice":40,
            "excTaxSellingPrice":50,
            "description": "exple",
            "stock": 20,
            "vatRate_id":1
    }
}'
```

Vous aurez une réponse **ok** du type :

```
{"message":"Product added with success"}
```

Et en cas d'erreurs comme un champ manquant ou une catégorie non trouvée :

```
{"message":"You have some errors in your request","errors":{"category_id":[
"Unable to find Category with id 10"]}
```

Ici, je vous présente l'idée générale de la *clean architecture*, comme je le disais plus haut, ce n'est pas un dogme à suivre aveuglément (« Telle est la Voie »), il faut vous l'approprier et déplacer le curseur pour en tirer le maximum de bénéfice avec le minimum de contraintes.

Pour revenir à notre exemple, on n'imagine pas devoir créer pour chaque action d'exécution une classe **response** unique ainsi qu'une interface unique et devoir implémenter son **presenter** unique, étant donné que toutes nos exécutions auraient le même type de retour.

On pourrait donc ici créer une réponse, une interface de présentation et générique, par exemple :

- **src/Domain/Response/ExecutionResponse.php** ;
- **src/Domain/Presenter/ExecutionPresenterInterface.php** ;
- **src/Infrastructure/Symfony/Presenter/ExecutionPresenterToJson.php**.

Pas besoin ici d'afficher cet exemple de code générique, c'est le même que précédemment avec un autre nom de classe et situé à la racine du répertoire type correspondant. Vous pouvez si besoin, comme pour chaque article, aller voir les fichiers génériques sur le dépôt GitHub associé.

7.2 Web

Passons à une implémentation Web pour voir la différence :

Créez l'arborescence :

```
mkdir -p Infrastructure/Symfony/Controller/Web
```

- Fichier **src/Infrastructure/Symfony/Controller/Web/ProductController.php** :

```
<?php

namespace Infrastructure\Symfony\Controller\Web;

use Domain\Entity\Product;
use Domain\Gateway\CategoryGatewayInterface;
use Domain\Gateway\ProductGatewayInterface;
use Domain\Gateway\VATRateGatewayInterface;
use Domain\Request\Sale\AddProductToSellRequest;
use Domain\Tool\Validator;
use Domain\UseCase\Sale\AddProductToSellUseCase;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\FormError;


class ProductController extends AbstractController{
```

```
    /**
     * @Route("/Web/product/add" )
     */
    public function addFormAction(
                    Request $httpRequest,
                    ProductGatewayInterface $productGateway,
                    VATRateGatewayInterface $vatRateGateway,
                    CategoryGatewayInterface $categoryGateway){

     $dropdownCategoryList=$categoryGateway->getDropdownList();
     $dropdownVatRateList=$vatRateGateway->getDropdownList();

     $form= $this->createFormBuilder(new Product())
     ->add('title', TextType::class)

     ->add('category_id',  ChoiceType::class,['choices'=>$dropdownCategoryLi
st])
      ->add('vatRate_id',  ChoiceType::class,['choices'=>$dropdownVatRateList
])

     ->add('excTaxBuyingPrice', TextType::class)
     ->add('excTaxSellingPrice', TextType::class)
     ->add('description', TextType::class)
     ->add('stock', TextType::class)


     ->add('save', SubmitType::class, ['label' => 'Create Product'])
     ->getForm();

     $form->handleRequest($httpRequest);

     if ($form->isSubmitted() && $form->isValid()) {

         $useCaseRequest=new AddProductToSellRequest();
         $useCaseRequest->setProductToAdd(new  Product(  (array)$form->getDat
a()));

         $useCase=new  AddProductToSellUseCase($productGateway,$vatRateGatew
ay,$categoryGateway,new Validator());
         $response=$useCase->execute($useCaseRequest);

         if($response->isStatusSuccess()){
             return $this->redirect('/Web/product/list');
         }

         $errorList=$response->getFieldErrorMessageList();

         if($response->isStatusException()){
            $form->addError(new  FormError(  $response->getException()->getMe
ssage() ));
         }else{
            foreach($errorList as $field => $errorMessage){
                $form->get($field)->addError(new  FormError(  join(',',$error
Message) ));
            }
         }
     }
```

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

  }

}
```

Cette fois, on utilise **twig** et le **form builder** pour le rendu, on détecte que le formulaire est envoyé pour appeler notre *use case*. En cas de réponse positive, on redirige sur la page listant les produits et en cas d'erreur (champ non valide...), on récupère la liste des erreurs pour les injecter au formulaire et lui laisser gérer l'affichage proprement.

Jetons juste un œil à la vue **twig** utilisée :

- Fichier **src/Infrastructure/Twig/templates/Sale/AddProductToSell.html.twig** :

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

{% block body %}

<div class="form">

{{ form(form) }}

</div>


{% endblock %}
```

Ceci est une simple vue/*template* où l'on étend notre *template* **twig** de base et l'on se contente d'afficher notre formulaire.

8. Changer le modèle de données

Passons à un cas concret de migration d'un élément de notre infrastructure. Passons notre mode de stockage de nos produits de nos fichiers plats à une base MariaDB.

8.1 Création de notre base de données MariaDB

Créez une base **expleMyShop** et ajoutez la table suivante :

```
CREATE TABLE `products` (
  `PR_PKEY` int(11) NOT NULL,
  `PR_Title` varchar(100) NOT NULL,
  `PR_Desc` text NOT NULL,
  `PR_excTaxBuyingPrice` decimal(10,0) NOT NULL,
  `PR_excTaxSellingPrice` decimal(10,0) NOT NULL,
  `PR_incTaxSellingPrice` decimal(10,0) NOT NULL,
  `PR_stock` int(11) NOT NULL,
```

```
  `PR_visible` int(11) NOT NULL,
  `PR_category_id` int(11) NOT NULL,
  `PR_vatRate_id` int(11) NOT NULL
)
```

Note : j'ai fait exprès d'avoir des noms de champs différents pour voir une migration plus réaliste.

8.2 Installation d'un ORM : Doctrine

Ajoutons le package Doctrine à notre projet avec la commande :

```
symfony composer require symfony/orm-pack
```

À l'installation de ce module, Symfony vous a créé un répertoire **src/Entity** et **src/Repository**, que nous allons déplacer dans la partie infrastructure :

```
mkdir src/Infrastructure/Doctrine
mv src/Entity src/Infrastructure/Doctrine
mv src/Repository src/Infrastructure/Doctrine
```

Mettons à jour notre *autoloader* pour prendre en compte cette nouvelle infrastructure.

- Éditez **config/packages/doctrine.yaml** :
  - Remplacez :

```
    orm:
        auto_generate_proxy_classes: true
        naming_strategy: doctrine.orm.naming_strategy.underscore_number_awa
re
        auto_mapping: true
        mappings:
            App:
                is_bundle: false
                dir: '%kernel.project_dir%/src/Entity'
                prefix: 'App\Entity'
                alias: App
```

  -
  - Par :

```
    orm:
        auto_generate_proxy_classes: true
        naming_strategy: doctrine.orm.naming_strategy.underscore_number_awa
re
        auto_mapping: true
        mappings:
            App:
                is_bundle: false
                dir: '%kernel.project_dir%/src/Infrastructure/Doctrine/Enti
ty'
                prefix: 'Infrastructure\Doctrine\Entity'
                alias: App
```

Cette modification faite, nous pouvons demander à Doctrine de créer à partir de notre table en base de données notre classe ORM *products*.

Avant ça, il nous faut indiquer dans notre fichier **.env** la chaîne de connexion à notre base MariaDB :

- Fichier **.env** :

```
DATABASE_URL="mysql://root:XXXXX@127.0.0.1:3306/expleShop?serverVersion=5.7
&charset=utf8mb4"
```

Maintenant, nous pouvons lancer la génération avec la commande suivante :

```
php bin/console doctrine:mapping:import
"Infrastructure\Doctrine\Entity" annotation -
path=src/Infrastructure/Doctrine/Entity
```

Comme vous pouvez le lire dans la commande, on indique bien le *namespace* ainsi que la destination spécifique pour nos entités ORM.

Notre classe a bien été générée, mais il nous manque les *getter*/*setter*, nous pouvons les ajouter automatiquement avec un nouveau module. Ajoutons-le avec la commande :

```
symfony composer require symfony/maker-bundle --dev
```

Et maintenant, demandons-lui de générer les *getter*/*setter* :

```
php bin/console make:entity --regenerate
Infrastructure\\Doctrine\\Entity
```

Regardons rapidement notre classe générée :

- Fichier **src/Infrastructure/Doctrine/Entity/Products.php** :

```
<?php

namespace Infrastructure\Doctrine\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* Products
*
* @ORM\Table(name="products")
* @ORM\Entity
*/
class Products
{

    public function __construct(array $dataList=[])
    {
        if(count($dataList)>0){
           foreach($dataList as $field => $value){
                if(property_exists($this,$field)){
```

```
                    $this->$field = $value;
                }
            }
        }
    }

    /**
     * @var int
     *
     * @ORM\Column(name="PR_PKEY", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="PR_Title", type="string", length=100, nullable=fal
se)
     */
    private $title;

    /**
     * @var string
     *
     * @ORM\Column(name="PR_Desc", type="text", length=65535, nullable=fals
e)
     */
    private $description;

    /**
     * @var string
     *
     * @ORM\Column(name="PR_excTaxBuyingPrice", type="decimal", precision=1
0, scale=0, nullable=false)
     */
    private $excTaxBuyingPrice;

    /**
     * @var string
     *
  ...

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getTitle(): ?string
    {
        return $this->title;
    }

  ...



}
```

Notez l'ajout d'un constructeur qui permettra de peupler un objet lors de l'utilisation de l'ORM pour enregistrer en base de données.

Créons maintenant la classe **repository** qui nous permettra de remplacer notre stockage fichier :

- Fichier **src/Infrastructure/Doctrine/Repository/ProductRepository.php** :

```
<?php

namespace Infrastructure\Doctrine\Repository;

use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;
use Domain\Entity\Product;
use Domain\Gateway\GatewayResponse;
use Domain\Gateway\ProductGatewayInterface;
use Infrastructure\Doctrine\Entity\Products;

class ProductRepository extends ServiceEntityRepository implements ProductG
atewayInterface
{

    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Products::class);
    }

    public function add(Product $newProduct): GatewayResponse
    {
        try{

            $products=new Products((array)$newProduct );

            $entityManager = $this->getEntityManager();
            $entityManager->persist($products);
            $entityManager->flush();

            $response=new GatewayResponse();
            $response->setStatusSuccess();
            return $response;

        }catch(\Exception $e){
            $response=new GatewayResponse();
            $response->setStatusFailed();
            $response->setFailedMessage($e->getMessage());
            return $response;
        }
    }
    public function findAll(): array
    {
        $conn = $this->getEntityManager()->getConnection();

        $sql = '
            SELECT
            `PR_PKEY` as id,
            `PR_Title` as title,
```

```
            `PR_Desc` as description,
            `PR_excTaxBuyingPrice` as excTaxBuyingPrice,
            `PR_excTaxSellingPrice` as _excTaxSellingPrice,
            `PR_incTaxSellingPrice` as incTaxSellingPrice,
            `PR_stock` as stock,
            `PR_visible` as visible,
            `PR_category_id` as category_id,
            `PR_vatRate_id` as vatRate_id

            FROM products p

            ORDER BY id ASC
            ';
        $stmt = $conn->prepare($sql);
        $resultSet = $stmt->executeQuery();

        $rowList = $resultSet->fetchAllAssociative();

        return $this->convertToProductList($rowList);
    }

    private function convertToProductList($rowList){
        $productList=[];
        foreach($rowList as $row){
            $productList[]=new Product($row);
        }

        return $productList;
    }


}
```

Ici, une simple implémentation de notre classe d'accès à nos produits. Notez que l'on passe par un retour sous forme de tableau associatif pour pouvoir retourner des objets métier et non des objets au sens de l'ORM.

```
public function add(Product $newProduct): GatewayResponse
    {
        try{

            $products=new Products((array)$newProduct );

            $entityManager = $this->getEntityManager();
            $entityManager->persist($products);
            $entityManager->flush();
```

Juste un petit focus sur cette méthode d'ajout, où l'on utilise ce constructeur précédemment ajouté pour charger les données envoyées de notre **`useCase`** pour pouvoir persister simplement avec Doctrine.

Maintenant si vous retournez sur votre page, vous aurez une erreur : en effet, Symfony se retrouve avec 2 classes qui implémentent la même interface, et il ne sait laquelle choisir.

Soit vous pouvez supprimer/déplacer votre ancienne implémentation, ou lui enlever l'implémentation de l'interface pour qu'elle ne soit plus identifiée.

Ou vous pouvez également explicitement indiquer quelle implémentation choisir pour cette interface en éditant le fichier services :

- Fichier **`config/services.yaml`** :

```
    Domain\Gateway\ProductGatewayInterface:  '@Infrastructure\Doctrine\Repos
itory\ProductRepository'
```

Ajoutez cette simple ligne à la fin pour lui indiquer de manière explicite laquelle des deux utiliser.

La simple désactivation de l'ancienne implémentation devrait être plus simple à gérer :

- Fichier **`src/Infrastructure/FileDb/ProductRepository.php`** :

```
<?php
namespace Infrastructure\FileDb;

use Domain\Entity\Product ;
use Domain\Gateway\GatewayResponse;
use Domain\Gateway\ProductGatewayInterface;

class ProductRepository //implements ProductGatewayInterface
{
```

9. One more thing

Il peut être difficile de respecter ou de vérifier que tout le monde respecte cette règle d'isoler totalement le code métier de l'infrastructure. Vous pouvez mettre en place un outil pour vérifier cette règle.

Cet outil *open source* s'appelle **`Deptrac`**, il est disponible à l'adresse https://github.com/qossmic/deptrac.

Vous pouvez l'installer sur votre projet en téléchargeant la dernière version du fichier **`deptrac.phar`** disponible à l'adresse https://github.com/qossmic/deptrac/releases.

Une fois téléchargé, déplaçons-le à la racine de notre projet :

```
mv /home/mika/Downloads/deptrac.phar .
```

Créons ensuite un fichier de configuration :

- Fichier **`deptrac.yaml`** :

```
parameters:
  paths:
    - ./src/
```

```
  exclude_files: ~

  layers:
    -
      name: Domain
      collectors:
        -
          type: className
          regex: .*Domain\\.*

  ruleset:
    Domain:
- Domain
```

Cette librairie permet d'identifier des **`layers`** : que l'on pourrait traduire par couches en définissant comment identifier les classes correspondantes, puis d'indiquer quelle couche a le droit d'appeler quelle autre couche.

Pour simplement vérifier qu'aucune classe métier n'appelle des classes extérieures, cette configuration suffit.

Et pour vérifier, vous pouvez lancer la commande suivante :

```
php deptrac.phar --report-uncovered
```

J'ai fait exprès ici d'ajouter un **`use`** extérieur pour vous montrer le type de rapport en cas de violation de la règle :

```
34/34 [...] 100%

----------- ----------
  Reason      Domain
-----------
  Uncovered     Domain\UseCase\Sale\AddProductToSellUseCase   has
uncovered                       dependency                        on
Symfony\Component\HttpFoundation\Request
              /home/mika/www/LM/CleanArchiSf/articleLMCleanArchit
ectureSymfony/src/Domain/UseCase/Sale/AddProductToSellUseCase.php
:16
---------------------

-------------------- -----
  Report
-------------------- -----
  Violations           0
  Skipped violations   0
  Uncovered            1
  Allowed              0
  Warnings             0
  Errors               0
-------------------- -----
```

On voit que dans la classe **`useCase`**, j'ai fait un appel à une classe du *framework*, ceci peut donc s'avérer très utile.

Mais cet outil va plus loin, vous pouvez être bien plus précis, par exemple avec la configuration suivante :

```
parameters:
  paths:
    - ./src/
  exclude_files: ~

  layers:
    -
      name: UseCase
      collectors:
        -
          type: className
          regex: Domain\\UseCase\\.*
    -
      name: Request
      collectors:
        -
          type: className
          regex: Domain\\Request\\.*
    -
      name: Response
      collectors:
        -
          type: className
          regex: Domain\\Response\\.*
    -
      name: Gateway
      collectors:
        -
          type: className
          regex: Domain\\Gateway\\.*
    -
      name: Entity
      collectors:
        -
          type: className
          regex: Domain\\Entity\\.*
    -
      name: Tool
      collectors:
        -
          type: className
          regex: Domain\\Tool\\.*


  ruleset:
    UseCase:
      - Request
      - Response
      - Gateway
      - Entity
      - Tool
    Gateway:
      - Entity
    Request:
      - Entity
```

On a une vision exhaustive des règles à suivre avec pour chaque couche la couche autorisée. Ceci peut être très pratique à ajouter dans votre pipeline Gitlab ou Jenkins pour lever une erreur en cas de non-respect de l'isolation.

Conclusion

Dans cet article, vous avez pu découvrir je l'espère les avantages de la *clean architecture*.

Je sais bien : cette architecture vous demande d'écrire davantage de code, c'est donc plus long à écrire, mais vous verrez, une fois que vous aurez essayé sur un projet, qu'il sera bien plus confortable d'y faire de la maintenance ou de le faire évoluer que si vous l'aviez fait en MVC ou autre architecture plus traditionnelle.

Je n'ai pas évoqué ici le CQRS (*Command and Request Responsabilty Segregation*) : même si pour ceux qui connaissent, l'utilisation d'une classe `request`/`response` et d'un `presenter` y fait penser, je pense que ça pourrait faire l'objet d'un article à part, avec notamment cette volonté de bien séparer les commandes d'écriture/d'altération de la donnée de ceux de lecture et récupération de celle-ci.

Au final, toutes ces méthodes de travail, et d'organisation de votre code, sont toujours plus contraignantes à l'écriture, mais pensez au début de cet article quand j'évoquais le livre de l'oncle Bob « Clean Code » : on passe plus de temps à lire du code qu'à l'écrire.

Pensez donc à votre code sur le long terme : pensez à votre « vous » dans quelques semaines/mois/années qui devra se replonger sur cette même application, sans plus du tout avoir en tête la manière dont elle a été développée.

Avec la *clean architecture*, l'architecture hexagonale, le *DDD*… le prochain à devoir intervenir sur votre code vous remerciera d'avoir pensé à lui.

Ne laissez pas le code d'aujourd'hui devenir le *legacy* de demain...

https://connect.ed-diamond.com/gnu-linux-magazine/glmf-257/faire-de-la-clean-architecture-avec-symfony