TROOPERS

UN VOYAGE DU LEGACY AU DDD

Livre Blanc édité par **Troopers**
4 rue des olivettes
44000 Nantes
Siret 538 811 944 00031
contact@troopers.email

Octobre 2022

Licence couverture :
https://pixabay.com/fr/photos/montagnes-homme-paysage-vue-4423621

SOMMAIRE

LEGACY TO DDD, DU CONCRET !

L'utilisation d'outils pour améliorer la façon de développer des projets informatiques est la clef pour obtenir des résultats adaptés aux différents besoins priorisés d'un projet : coût, maintenabilité, rapidité, lisibilité, performance, évolutivité ...

Parmi eux, le **Domain Driven Development** (que nous nommerons DDD) améliore considérablement la qualité, la lisibilité et la testabilité sur le long terme, au détriment du temps de mise en place. Le postulat de base du DDD est simple : toute programmation devrait refléter la réalité du domaine (ou métier) avant toute considération technique.

Que faire si on réalise après coup que cette technique aurait sûrement été précieuse pour un projet ? Le passage d'un projet legacy en DDD est-il si coûteux ? Voyons cela ensemble par un exemple concret. L'idée est ici est de partir d'un projet existant, d'y apporter les modifications, en pointant le point de vue DDD.

Disons qu'un client vient me demander de moderniser une application existante en php 7.1 (dernière release le 24 octobre 2019), basée sur le framework Cakephp en version 3.1. Prenons en exemple la démo de Cakephp : https://github.com/cakephp/bookmarker-tutorial. Elle n'a pas été touchée depuis 2017. Je l'ai forkée pour l'occasion ici : https://github.com/vibby/cakephp-bookmarker-tutorial

> **DDD** Attention : Appliquer le DDD à cette application n'est sûrement pas très pertinent. Elle ne présente pas de forte valeur métier. Mais elle a l'avantage d'être facile à comprendre et disponible en open-source ^^ Parfait pour un exemple !

Par souci de lisibilité, j'ai systématiquement placé les propriétés en public dans les différentes classes. À ne pas faire sur un vrai projet !

Notre premier objectif sera d'identifier et d'extraire toutes les actions métier et de les mettre à part dans la toute nouvelle couche domaine de cette application. Ensuite, nous aborderons le chemin de la modernisation, et même celui de l'extension.

1. PRÉPARER LE TERRAIN

1.1. Faire fonctionner l'appli existante

Quand on a récupéré ce code, on va tâcher de le faire tourner en local. On peut voir qu'il y a de la configuration Ansible et Vagrant dessus. Mais comme nous sommes joueurs, je vous propose plutôt de passer par un petit docker-compose pour mettre ça en place ;)

```
# ./docker-compose.yaml
version: '3'

services:
  web:
    build:
      context: .docker/php
    networks:
      - backend
    depends_on:
      - db
    volumes:
      - .:/app:rw,cached
    ports:
      - 9050:80
    entrypoint: /docker-entrypoint.d/start.sh

  db:
    image: mysql:5
    networks:
      - backend
    volumes:
      - ./config/schema:/dumps:ro,cached
    environment:
      - DEBUG=false
      - MYSQL_USER=bookmark
      - MYSQL_PASSWORD=bookmark
      - MYSQL_DATABASE=bookmark
```

```
      - MYSQL_ROOT_PASSWORD=bookmark

networks:
  backend:
```

Je vous épargne la conf de tout ceci, mais vous trouverez tout dans ce commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/a4f354a2a6a049eac909ec9ee0008909f1316f0a

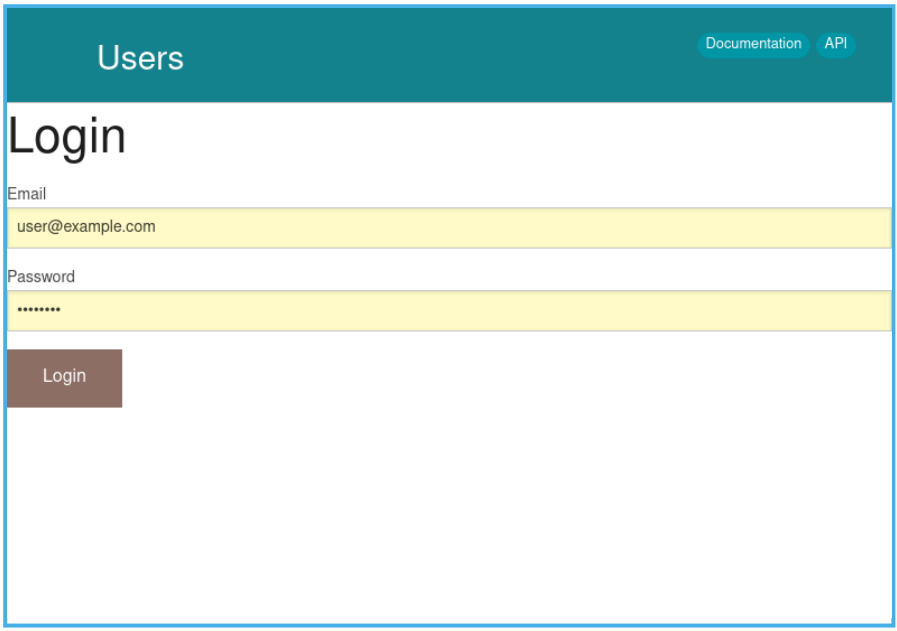
On lance le tout.

```
docker-compose up
```

On créé et on rempli la base de données

```
docker-compose exec db bash
mysql -u root -pbookmark
create database bookmark;
CREATE USER 'bookmark'@'%' IDENTIFIED BY 'bookmark';
GRANT ALL PRIVILEGES ON bookmark.* TO 'bookmark'@'%';
FLUSH PRIVILEGES;
exit
mysql -u root -pbookmark < /dumps/app.sql bookmark
mysql -u root -pbookmark < /dumps/i18n.sql bookmark
mysql -u root -pbookmark < /dumps/sessions.sql bookmark
exit
```

On peut déjà se connecter sur http://0.0.0.0:9050/users/login avec user@example.com et «password».



Pour chaque section, vous trouverez le parcours sur un historique git. En voici la première étape : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/a4f354a2a6a049eac909ec9ee0008909f1316f0a

1.2. Laisser de la place à nos nouveaux outils

La prochaine tâche consistera à faire de la place parmi les sources du projet pour de nouveau éléments. On va donc dédier un espace du dossier ./src pour les sources pré-existante.

```
mv src cakephp
mkdir src
mv cakephp ./src/
```

Puis on va modifier la résolution des domaines de noms pour qu'il trouve ces affaires au bon endroit.

```
# ./composer.json
…
    "autoload": {
        "psr-4": {
            "App\\": "src"
            "App\\": "src/cakephp"
        }
    },
…
```

Nous allons conserver le domaine de nom *App* pour la partie Cakephp, pour ne pas avoir a le modifier partout dans l'existant.

À noter qu'il faut aussi préciser à Cakephp où trouver ses templates et ses traductions.

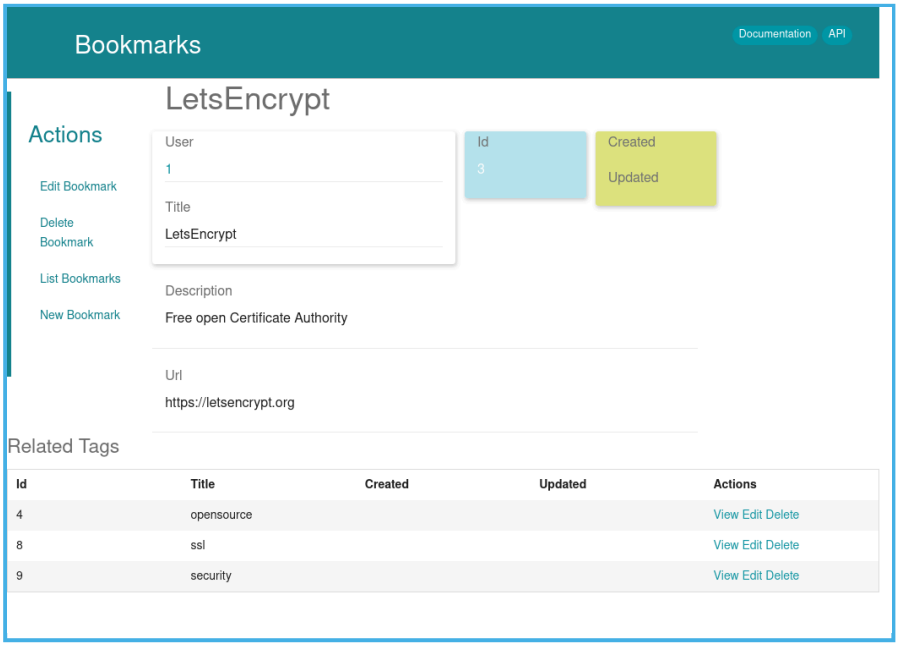
```
// config/app.php
…
        'paths' => [
            'plugins' => [ROOT . DS . 'plugins' . DS],
            'templates' => [APP . 'cakephp/Template' . DS],
            'locales' => [APP . 'cakephp/Locale' . DS],
        ],
…
```

Le commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/7121450d3f51d802714368ab93d4390b0ff1080a

2. COMMENCER PAR UN BOUT : AFFICHER UN BOOKMARK

Commençons pas un bout, le contrôleur **BookmarksController::view**. Vraisemblablement, il sert à afficher un bookmark, il faut donc le retrouver. Confions cette responsabilité à la couche métier !

Avant de modifier le contrôleur, créons le parcours de l'action coté domaine.



2.1. DDD : C'est parti !

En DDD, tout action faite au métier entre par une «porte» bien identifiée

: l'« entrypoint ». Il se présente sous la forme d'un objet qui explicite l'intention métier. Créons cet objet pour l'action de retrouver un bookmark.

Cet entrypoint ce place dans la couche « application » qui sert à faire l'interface entre l'infrastructure (cakephp dans notre cas), et la couche du domaine (ou métier).

```
namespace Application\GetBookmark;

class GetBookmarkInput
{
    public $id;
}
```

Puis nous créons un *handler* capable de la traiter.

```
use \Domain\Bookmark\Repository\BookmarkRepository;
use \Domain\Bookmark\Model\Bookmark;

class GetBookmarkHandler
{
    private $bookmarkRepository;

    public function __construct(
        BookmarkRepository $bookmarkRepository
    ) {
        $this->bookmarkRepository = $bookmarkRepository;
    }

    public function __invoke(
        GetBookmarkInput $input
    ): ?Bookmark {
        return $this->bookmarkRepository->find($input->id);
    }
}
```

Maintenant, le repository pour aller chercher la donnée.

```
namespace Domain\Bookmark\Repository;

use \Domain\Bookmark\Model\Bookmark;

interface BookmarkRepository
```

```
{
    public function find(string $id): ?Bookmark;
}
```

Vous remarquerez que ce n'est qu'une simple interface, nous mettrons son implémentation sur pied tout à l'heure, côté Cakephp.

Et finalement, voici enfin l'objet métier !

```
namespace Domain\Bookmark\Model;

class Bookmark
{
    public $id;
    public $title;
    public $url;
    public $description;
}
```

OK, je l'ai simplifié au max pour vous faciliter la lecture :) À ne pas faire sun une vraie prod !

N'oublions pas de faire connaître cette nouvelle partie à l'autoloader.

```
# ./composer.json
…
    "autoload": {
        "psr-4": {
            "App\\": "src/cakephp",
            "Application\\": "src/Application",
            "Domain\\": "src/Domain"
        }
    },
…
```

Puis mettre à jour ses résolutions.

```
docker-compose exec web composer dump-autoload
```

Résumé : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/064c6a84ac65753746ad9ae273b056857402c033

DDD Cette action est très simple, elle répond au besoin rudimentaire d'un affichage. En DDD, c'est le domaine qui décide de ce points d'entrées. Il n'est pas question ici de faire un simple CRUD. Mais un point d'entrée à un véritable sens pour le métier. Ce qui est tout à fait possible pour ce point de lecture d'un bookmark.

2.2. Appel depuis le contrôleur Cakephp

Quand on essaye de faire appel à la couche métier depuis Cakephp, on se heurte à un problème : Cakephp n'a pas de mécanisme de container pour gérer les dépendances entre les classes. Qu'à cela ne tienne, nous allons créer notre propre container sur mesure. Utilisons pour cela le concept de composant de Cakephp

```
namespace App\Controller\Component;

use Application\GetBookmark\GetBookmarkHandler;
use Cake\Controller\Component;
use Cake\Controller\ComponentRegistry;
use Exception;

/**
 * @property BookmarkRepositoryComponent $BookmarkRepository
 */
class ContainerComponent extends Component
{
    public $components = ['BookmarkRepository']; // C'est la magie de Cakephp
pour charger un composant depuis un autre composant
    private $container = [];

    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        parent::__construct($registry, $config);

        $this->container[GetBookmarkHandler::class] = new
GetBookmarkHandler($this->BookmarkRepository); // On crée nos services ici
    }
```

```
    public function get($serviceName)
    {
        if (!isset($this->container[$serviceName])) {
            throw new Exception('Cannot find service');
        }

        return $this->container[$serviceName];
    }
}
```

Définir chaque service individuellement sera sûrement un peu fastidieux, mais c'est le prix a payer pour faire cohabiter des paradigmes aussi éloignés que la philosophie de Cakephp et l'injection de dépendance.

En paramètre du constructeur de *GetBookmarkHandler*, nous avons donné le composant *BookmarkRepositoryComponent*. Il doit implémenter l'interface BookmarkRepository.

```
namespace App\Controller\Component;

use App\Model\Entity\Bookmark;
use App\Model\Table\BookmarksTable;
use Cake\Controller\Component;
use Cake\ORM\TableRegistry;
use Domain\Bookmark\Model\Bookmark as BookmarkModel;
use Domain\Bookmark\Repository\BookmarkRepository;

/**
 * @property BookmarksTable $Bookmarks
 */
class BookmarkRepositoryComponent extends Component implements
BookmarkRepository
{
    public function findById(string $id): ?BookmarkModel
    {
        $Bookmarks = TableRegistry::get('Bookmarks'); // C'est la magie de
Cakephp pour récupérer l'objet table qui fait les requêtes
        $bookmarkEntity = $Bookmarks->get($id);
        if (!$bookmarkEntity instanceof Bookmark) {
            return null;
        }
        $bookmarkModel = new BookmarkModel();
        $bookmarkModel->id = $bookmarkEntity->id;
        $bookmarkModel->title = $bookmarkEntity->get('title');
```

```
        $bookmarkModel->url = $bookmarkEntity->get('url');
        $bookmarkModel->description = $bookmarkEntity->get('description');

        return $bookmarkModel;
    }
}
```

Dans la dernière partie, on construit l'objet métier à partir de l'entité Cakephp, car c'est le contrat exigé par la couche domaine.

Maintenant, il nous faut importer le composant Container dans la fonction initialize du contrôleur.

```
    $this->loadComponent('Container');
```

Et nous pouvons nous en servir dans le contrôleur.

```
namespace App\Controller;

use \Application\GetBookmark\GetBookmarkInput;

class BookmarksController extends AppController
{
    …

    /**
     * View method
     *
     * @param string|null $id Bookmark id.
     * @return void
     * @throws \Cake\Network\Exception\NotFoundException When record not found.
     */
    public function view($id = null)
    {
        $input = new GetBookmarkInput();
        $input->id = $id;
        $handler = $this->Container->get(GetBookmarkHandler::class);
        $bookmarkModel = $handler($input);
        $bookmark= new Bookmark();
        $bookmark->set('title', $bookmarkModel->title);
        $bookmark->set('url', $bookmarkModel->url);
        $bookmark->set('description', $bookmarkModel->description);
        $bookmark->set('id', $bookmarkModel->id);
```

```
        $this->set('bookmark', $bookmark);
        $this->set('_serialize', ['bookmark']);
    }
    …
}
```

Là, on vient de faire exactement le chemin inverse qu'auparavant : passer de l'objet métier à l'entité Cakephp, puisque c'est sa façon de travailler.

On peut constater que la partie Cakephp ne remplit aucune autre fonction que l'interprétation de la requête HTTP et le rendu d'un retour. C'est précisément ce que l'on attend de la couche infrastructure. Les actions métiers (sur les données) sont déléguées à la couche domaine.

Les modifications : https://github.com/cakephp/bookmarker-tutorial/commit/33595707c4f7e2dc3cf09b064fe2d6710129ac9c

Et voilà ! Notre application passe maintenant par une couche domaine indépendante pour manipuler nos données ! Il n'y a pour le moment aucun intérêt, on a même perdu des fonctionnalités :) Mais progressons dans cette voie !

2.3. Iso-fonctionnel

Sur le site de l'application, à la page d'un bookmark, on peut voir qu'il manque deux éléments importants : l'utilisateur «propriétaire» et les tags associés. Il va nous falloir les ajouter en tant que modèles.

```
namespace Domain\Bookmark\Model;

class Tag
{
    public $id;
    public $title;
    public $bookmarks;
}
```

```
namespace Domain\Bookmark\Model;

class User
```

```
{
    public $id;
    public $email;
    public $password;
    public $dateOfBirth;
    public $bookmarks;
}
```

Et ajoutons ces propriétés au bookmark

```
namespace Domain\Bookmark\Model;

class Bookmark
{
    …
    public $user;
    public $tags;
}
```

Malheureusement, nous ne pouvons pas typer ces propriétés : cette possibilité n'existe que depuis PHP 7.4. Nous reviendrons dessus.

Il nous reste a demander à l'ORM de Cakephp d'aller chercher ces données et de les hydrater.

```
namespace App\Model\Table;

…
use Domain\Bookmark\Model\Tag;
use Domain\Bookmark\Model\User;
…

class BookmarkRepositoryComponent extends Component implements
BookmarkRepository
{
    public function findById(string $id): ?BookmarkModel
    {
        …
        $bookmarkEntity = $this->get($id);
        $bookmarkEntity = $this->get($id, [
            'contain' => ['Users', 'Tags']
        ]);
        …
```

```
        $userModel = new User();
        $userModel->id = $bookmarkEntity->get('user')->get('id');
        $userModel->email = $bookmarkEntity->get('user')->get('email');
        $userModel->dateOfBirth = $bookmarkEntity->get('user')->get('dob');
        $bookmarkModel->user = $userModel;

        foreach ($bookmarkEntity->get('tags') as $tag) {
            $tagModel = new Tag();
            $tagModel->id = $tag->get('id');
            $tagModel->title = $tag->get('title');
            $bookmarkModel->tags[] = $tagModel;
        }
        …
    }
}
```

Et le chemin retour, quand on revient au contrôleur.

```
namespace App\Controller;

…
use App\Model\Entity\Tag;
use App\Model\Entity\User;
…
class BookmarksController extends AppController
{

    public function view($id = null)
    {
        …
        $user = new User();
        $user->set('id', $bookmarkModel->user->id);
        $user->set('email', $bookmarkModel->user->email);
        $bookmark->set('user', $user);

        $tags = [];
        foreach ($bookmarkModel->tags as $tagModel) {
            $tag = new Tag();
            $tag->set('id', $tagModel->id);
            $tag->set('title', $tagModel->title);
            $tags[] = $tag;
        }
        $bookmark->set('tags', $tags);
```

```
        …
    }
}
```

Nos tags et notre utilisateur s'affiche maintenant correctement.

Par ici le commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/5f592d8bbc02a4e709d2184a7d1822447008d3f8

2.4. Petite refactorisation

Nous avons finalement deux systèmes pour représenter les mêmes données. Il y a un objet bookmark côté domaine (Model), mais également côté Cakephp (Entity).

Avec un peu d'expérience, on sent bien qu'on va passer beaucoup de temps à transformer nos données d'un système à l'autre, alors outillons nous pour ce boulot !

```php
namespace App\Controller\Component;

use App\Model\Entity\Bookmark;
use Cake\Controller\Component;
use Domain\Bookmark\Model\Bookmark as BookmarkModel;

/**
 * @property TagTransformerComponent $TagTransformer
 * @property UserTransformerComponent $UserTransformer
 */
class BookmarkTransformerComponent extends Component
{
    public $components = ['TagTransformer', 'UserTransformer'];

    public function modelToEntity(BookmarkModel $bookmarkModel): Bookmark
    {
        $bookmarkEntity= new Bookmark();
        $bookmarkEntity->set('title', $bookmarkModel->title);
        $bookmarkEntity->set('url', $bookmarkModel->url);
        $bookmarkEntity->set('description', $bookmarkModel->description);
        $bookmarkEntity->set('id', $bookmarkModel->id);

        if ($bookmarkModel->user) {
```

```
            $bookmarkModel->user =
$this->UserTransformer->entityToModel($bookmarkEntity->get('user'));
        }
        if ($bookmarkModel->tags) {
            $bookmarkEntity->set('tags', array_map(
                function ($tagModel) {
                    return $this->TagTransformer->modelToEntity($tagModel);
                },
                $bookmarkModel->tags
            ));
        }

        return $bookmarkEntity;
    }

    public function EntityToModel(Bookmark $bookmarkEntity): BookmarkModel
    {
        $bookmarkModel = new BookmarkModel();
        $bookmarkModel->id = $bookmarkEntity->id;
        $bookmarkModel->title = $bookmarkEntity->get('title');
        $bookmarkModel->url = $bookmarkEntity->get('url');
        $bookmarkModel->description = $bookmarkEntity->get('description');

        if ($bookmarkEntity->get('user')) {
            $bookmarkEntity->set('user',
$this->UserTransformer->entityToModel($bookmarkEntity->get('user')));
        }
        if ($bookmarkEntity->get('tags')) {
            $bookmarkModel->tags = array_map(
                function ($tagEntity) {
                    return $this->TagTransformer->entityToModel($tagEntity);
                },
                $bookmarkEntity->get('tags')
            );
        }

        return $bookmarkModel;
    }
}
```

La profondeur de transformation de la donnée sera directement liée au données que l'on a de présente au départ. Par exemple, si l'utilisateur n'est pas fourni au départ, il restera vide. Ici, j'ai décidé de séparer les différentes entités dans des transformateurs différents. Voici celui d'un tag.

```php
namespace App\Controller\Component;

use Cake\Controller\Component;
use Domain\Bookmark\Model\Tag as TagModel;
use App\Model\Entity\Tag as TagEntity;

class TagTransformerComponent extends Component
{
    public function modelToEntity(TagModel $tagModel): TagEntity
    {
        $tagEntity= new TagEntity();
        $tagEntity->set('title', $tagModel->title);
        $tagEntity->set('id', $tagModel->id);

        return $tagEntity;
    }

    public function EntityToModel(TagEntity $tagEntity): TagModel
    {
        $tagModel = new TagModel();
        $tagModel->id = $tagEntity->id;
        $tagModel->title = $tagEntity->get('title');

        return $tagModel;
    }
}
```

Et enfin l'utilisateur

```php
namespace App\Controller\Component;

use App\Model\Entity\User;
use App\Model\Entity\User as UserEntity;
use Cake\Controller\Component;
use Domain\Bookmark\Model\User as UserModel;

class UserTransformerComponent extends Component
{
    public function modelToEntity(UserModel $userModel): UserEntity
    {
        $userEntity = new UserEntity();
        $userEntity->set('id', $userModel->id);
        $userEntity->set('email', $userModel->email);
```

```
        return $userEntity;
    }

    public function EntityToModel(UserEntity $userEntity): UserModel
    {
        $userModel = new UserModel();
        $userModel->id = $userEntity->id;
        $userModel->email = $userEntity->get('email');

        return $userModel;
    }
}
```

Il nous reste maintenant à nous en servir dans les parties concernées : le contrôleur et le repository

```
namespace App\Controller;
…
class BookmarksController extends AppController
{
    …
    public function view($id = null)
    {
        $input = new GetBookmarkInput();
        $input->id = $id;
        $handler = $this->Container->get(GetBookmarkHandler::class);
        $bookmarkModel = $handler($input);
        $bookmark = $this->BookmarkTransformer->modelToEntity($bookmarkModel);

        $this->set('bookmark', $bookmark);
        $this->set('_serialize', ['bookmark']);
    }
    …
}
```

```
namespace App\Controller\Component;
…
class BookmarkRepositoryComponent extends Component implements
BookmarkRepository
{
    public $components = ['BookmarkTransformer'];

    public function findById(string $id): ?BookmarkModel
    {
```

```
        $Bookmarks = TableRegistry::get('Bookmarks');
        $bookmarkEntity = $Bookmarks->get($id, [
            'contain' => ['Users', 'Tags']
        ]);
        if (!$bookmarkEntity instanceof Bookmark) {
            return null;
        }
        return $this->BookmarkTransformer->EntityToModel($bookmarkEntity);
    }

}
```

Voilà qui simplifie les choses et les rend plus lisibles.

Retrouvez le détail de ces modifications ici : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/b60acf7e0dd636b3f290c1f68826ee1c8043c6a6

2.5. Point d'étape

Nous avons donc le même résultat fonctionnel, tout en passant par notre couche domaine. Ce qui signifie que pour l'utilisateur, la refonte est parfaitement transparente ! Il est donc tout à fait possible de fusionner ces modifications au reste de la base de code, et d'envoyer tout ça en production.

Ce que nous sommes en train de faire là, c'est de la refonte en continu. Sans interruption de service, sans avoir à passer 1 ou 2 ans dans une refonte avant de pouvoir livrer. Pour un décideur ou un chef de projet, il y a là une grande valeur.

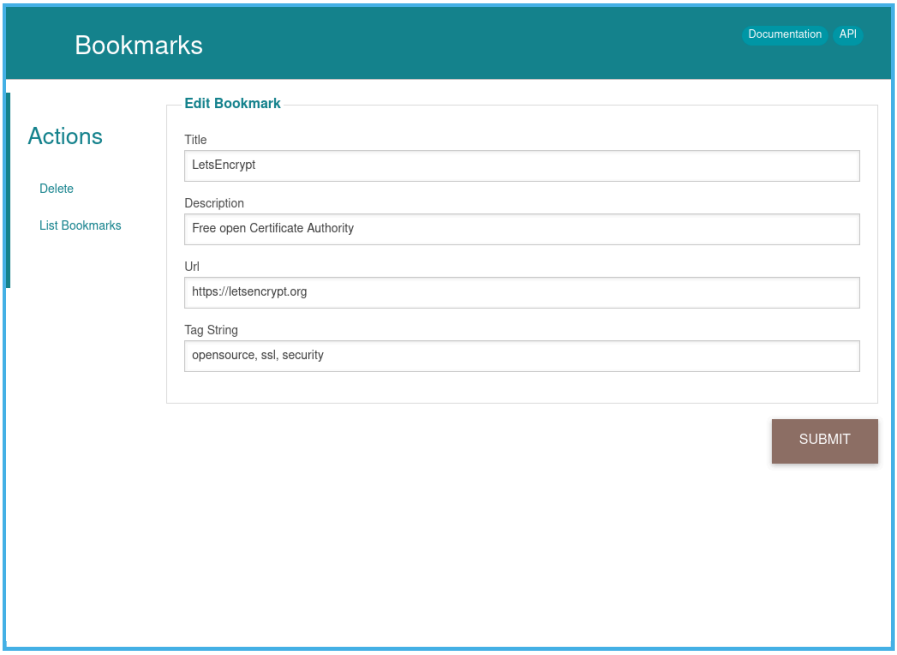
Dans notre cas, nous avons à faire à une simple mise à jour des données, c'est pourquoi la porter dans la couche domaine n'a pas été complexe, et aussi qu'elle n'a apporté aucune plus-value. Mais pour une application qui mérite de passer au DDD, le processus de porter les règles métier à la couche domaine permet de les identifier clairement, de les faire comprendre et valider par l'équipe, et de les modifier simplement si besoin. Pour être plus concret, voici quelques exemple de règle métier complexes :

- Une réduction sur un e-commerce qui s'applique sur les produits d'une catégorie à condition d'atteindre une certaine somme

dans cette catégorie, et uniquement pour les clients n'ayant jamais commandé dans cette catégorie
- Une règle comptable pour définir si la TVA peut être décomptée ou non d'une facture
- Un retour d'erreur d'un diagnostic OBD sur un véhicule automobile

3. MODIFIER UN BOOKMARK

Une opération de lecture a été faite. Mais une opération d'écriture sera peut-être un peu plus complexe ? Attaquons-nous à la modification d'un bookmark.



3.1. Commencer par le domaine métier

À nouveau, on va commencer par implémenter le domaine métier.

DDD Cela n'a rien d'un choix éditorial, mais ça correspond bien à la philosophie DDD : le domaine métier doit être au cœur de l'application. Elle est prioritaire et doit se suffire à elle-même. C'est à la partie infrastructure de s'adapter pour adhérer au domaine.

Créons donc l'input puis le handler.

```php
namespace Application\GetBookmark;

class UpdateBookmarkInput
{
    public $id;
    public $title;
    public $url;
    public $description;
}
```

```php
namespace Application\GetBookmark;

use Domain\Bookmark\Repository\BookmarkRepository;
use Domain\Bookmark\Model\Bookmark;
use Domain\Bookmark\Updater\BookmarkUpdater;

class UpdateBookmarkHandler
{
    private $bookmarkRepository;
    private $updater;

    public function __construct(
        BookmarkRepository $bookmarkRepository,
        BookmarkUpdater $updater
    ) {
        $this->bookmarkRepository = $bookmarkRepository;
        $this->updater = $updater;
    }

    public function __invoke(
        UpdateBookmarkInput $input
    ): ?Bookmark {
        $bookmark = $this->bookmarkRepository->findById($input->id);
        if (!$bookmark) {
            return null;
        }
```

```
        $bookmark = $this->updater->update(
            $bookmark,
            $input->title,
            $input->url,
            $input->description,
        );

        if ($bookmark) {
            $this->bookmarkRepository->persist($bookmark);
        }

        return $bookmark;
    }
}
```

On fait ici appel à un updater qui va véritablement appliquer les modifications au bookmark.

```
namespace Domain\Bookmark\Updater;

use Domain\Bookmark\Model\Bookmark;

class BookmarkUpdater
{
    public function update(
        Bookmark $bookmark,
        string $title,
        string $url,
        string $description
    ): ?Bookmark {
        $bookmark->title = $title;
        $bookmark->url = $url;
        $bookmark->description = $description;

        return $bookmark;
    }
}
```

Son utilisation peut sembler superflu pour le moment, mais il se rendra très utile prochainement.

Il nous faut mettre en place la mise à jour des données. Nous le faisons dans le repository pas simplicité. Mais il serait pertinent d'avoir une autre interface pour les écritures (`Domain\Bookmark\Depository\`

BookmarkDepository)

```
namespace Domain\Bookmark\Repository;

use \Domain\Bookmark\Model\Bookmark;

interface BookmarkRepository
{
    …
    public function persist(Bookmark $bookmark): void;
}
```

Petit commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/f4aea58629c5dc793e9e241a2ab6c4fc55243d88

3.2. La mise à jour côté infra

Comme nous l'avons fait pour la vue, on va faire appel à la couche métier pour les interactions avec les données

```
namespace App\Controller;
…
use Application\UpdateBookmark\UpdateBookmarkHandler;
use Application\UpdateBookmark\UpdateBookmarkInput;
…
class BookmarksController extends AppController
{
    public function edit($id = null)
    {
        if ($this->request->is(['patch', 'post', 'put'])) {
            $input = new UpdateBookmarkInput();
            $input->id = $id;
            $input->title = $this->request->data['title'];
            $input->url = $this->request->data['url'];
            $input->description = $this->request->data['description'];
            $handler = $this->Container->get(UpdateBookmarkHandler::class);
            $bookmarkModel = $handler($input);
            if ($bookmarkModel) {
                $this->Flash->success('The bookmark has been saved.');
                return $this->redirect(['action' => 'index']);
            }
            $bookmark =
```

```
$this->BookmarkTransformer->modelToEntity($bookmarkModel);
            $this->Flash->error('The bookmark could not be saved. Please, try
again.');
        } else {
            $input = new GetBookmarkInput();
            $input->id = $id;
            $handler = $this->Container->get(GetBookmarkHandler::class);
            $bookmarkModel = $handler($input);
            $bookmark =
$this->BookmarkTransformer->modelToEntity($bookmarkModel);
        }
        …
    }

}
```

Il nous faut implémenter aussi la mise à jour des données.

```
namespace App\Controller\Component;
…
class BookmarkRepositoryComponent extends Component implements
BookmarkRepository
{
    …
    public function persist(BookmarkModel $bookmark): void
    {
        $Bookmarks = TableRegistry::get('Bookmarks');
        $Bookmarks->save($this->BookmarkTransformer->ModelToEntity($bookmark));
    }
}
```

Nous avons bien fait de mettre à part notre système de transformation des objet du système Cakephp au domaine métier, car nous l'avons utilisé pas moins de 5 fois déjà !

Nous avons créé de nouveau services qu'il nous faut préciser au container.

```
namespace App\Controller\Component;

use Application\UpdateBookmark\UpdateBookmarkHandler;
use Domain\Bookmark\Updater\BookmarkUpdater;
…
```

```
class ContainerComponent extends Component
{
    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        …
        $this->container[GetBookmarkHandler::class] = new
GetBookmarkHandler($this->BookmarkRepository); // On crée nos services ici
        $this->container[BookmarkUpdater::class] = new BookmarkUpdater();
        $this->container[UpdateBookmarkHandler::class] = new
UpdateBookmarkHandler($this->BookmarkRepository,
$this->container[BookmarkUpdater::class]);
    }
    …
}
```

Le commit de cette partie : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/355bc6e08aa9cc9d21116032be85a0956b433b1c

À présent, nous avons une interface opérationnelle pour la modification d'un bookmark, mais il manque une fonctionnalité ;)

3.3. Les entités liées

Nous avons traité le cas simple ici, mais nous avons écarté la liaison entre bookmark et tag. Il y a ici une spécificité métier qu'il va nous falloir implémenter : la liste des tags est donnée sous la forme d'une liste de titres. Quand un titre inexistant parmi les tags est trouvé, il faut créer ce tag. C'est une action qui est faite pour le moment dans la classe App\Model\Table\BookmarksTable

Nous allons mettre à profit notre BookmarkUpdater pour y matérialiser cette logique.

```
namespace Domain\Bookmark\Updater;

…
use Domain\Bookmark\Model\Tag;
use Domain\Bookmark\Repository\TagRepository;
```

```php
class BookmarkUpdater
{
    private $tagRepository;

    public function __construct(
        TagRepository $tagRepository
    ){
        $this->tagRepository = $tagRepository;
    }

    public function update(
        …
        array $tagsTitle
    ): ?Bookmark {
        …
        $bookmark->tags = [];
        foreach ($tagsTitle as $tagTitle) {
            $tag = $this->tagRepository->findByTitle($tagTitle);
            if (!$tag) {
                $tag = new Tag();
                $tag->title = $tagTitle;
            }
            $bookmark->tags[] = $tag;
        }

        return $bookmark;
    }
}
```

Ce code n'est pas optimisé, puisqu'on fait autant d'appels à la base de données que de titres de tag. Je l'ai fait ainsi pour en faciliter la lecture.

Nous avons fait appel à un nouveau repository ici.

```php
namespace Domain\Bookmark\Repository;

interface TagRepository
{
    public function findByTitle(string $title): ?Tag;
}
```

En remontant la chaîne, on ajoute ce paramètre au handler.

```
namespace Application\UpdateBookmark;

class UpdateBookmarkHandler
{
    public function __invoke(UpdateBookmarkInput $input)
    {
        …
        $bookmark = $this->updater->update(
            …
            $input->tagsTitle
        );
        …
    }
}
```

Puis au tour de l'input

```
namespace Application\UpdateBookmark;

class UpdateBookmarkInput
{
    …
    public $tagsTitle;
}
```

En résumé ici : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/4ecad97fb7f82f7951bc956cda763172484c0047

3.4. Tags sur la couche infra

Côté contrôleur, il nous faut préciser la liste des titres de tags passés en paramètre.

```
namespace App\Controller;

class BookmarksController extends AppController
{
    …
    public function edit($id = null)
    {
        if ($this->request->is(['patch', 'post', 'put'])) {
```

```
            …
            $input->tagsTitle = array_filter(
                array_unique(
                    array_map(
                        'trim',
                        explode(',', $this->request->data['tag_string'])
                    )
                )
            );
            …
        }
        …
    }
}
```

Et nous avons introduit un tout nouveau TagRepository que nous implémentons maintenant

```
namespace App\Controller\Component;

use App\Model\Table\BookmarksTable;
use Cake\Controller\Component;
use Cake\ORM\TableRegistry;
use Domain\Bookmark\Model\Tag;
use Domain\Bookmark\Repository\TagRepository;

/**
 * @property BookmarksTable $Bookmarks
 * @property TagTransformerComponent TagTransformer
 */
class TagRepositoryComponent extends Component implements TagRepository
{
    public $components = ['TagTransformer'];

    public function findByTitle(string $title): ?Tag
    {
        $Tags = TableRegistry::get('Tags');
        $tagEntity = $Tags->find()->where(['Tags.title =' => $title])->first();
        if (!$tagEntity) {
            return null;
        }
```

```
        return $this->TagTransformer->EntityToModel($tagEntity);
    }
}
```

Pour parachever le tout, ne pas oublier de mettre à jour notre service avec sa nouvelle dépendance.

```
namespace App\Controller\Component;
…
class ContainerComponent extends Component
{
    …
    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        …
        $this->container[BookmarkUpdater::class] = new BookmarkUpdater();
        $this->container[BookmarkUpdater::class] = new
BookmarkUpdater($this->TagRepository);
    }
}
```

Et puisque nous l'avons transféré côté métier, nous pouvons débrancher la fonctionnalité côté Cakephp en supprimant les méthodes `beforeSave` et `_buildTags` de la classe `App\Model\Table\BookmarksTable`.

En voici le résumé : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/4184cb19417b389b069ad941040aa8a5e2f314ef

Nous avons maintenant un cas complet d'écriture en passant par la couche du domaine. Comme pour la vue, nous sommes iso-fonctionnel avec l'existant. Nous pouvons pousser en prod ^^

3.5. Point d'étape

Nous avons vu deux cas de base d'une application web : récupération d'une donnée et modification d'une donnée. Sur la base de cette approche, toute action pourra être maintenant réalisée. Et toute règle métier pourra être implémentée du côté domaine. Nous voici parés d'un bon pied pour moderniser cette petite application.

Nous pourrions maintenant passer en revue toutes les actions des contrôleurs pour leur appliquer le même sort. Mais je vous laisse cela en travaux pratiques :) Les MR sont les bienvenues sur le repo.

DDD Si vous avez surement remarquer que les dépendances ne se font que dans un seul sens : l'infrastructure utilise le domaine, mais le domaine reste bien indépendante la couche infrastructure.

Il nous restera cependant quelques épines. Il restera côté Cakephp des règles métier qui devraient trouver leur place dans le domaine. Je pense notamment à la validation des entités. Elle passe aujourd'hui par la classe `BookmarkTable`, notamment à travers la méthode `validationDefault`. Il nous faudra bien transférer ces règles dans le domaine. Et d'une manière générale, il faudra dépouiller toutes les classe finissant par `Table`, à l'exception de la méthode `initialize`.

Un autre point tout à fait remarquable aussi : Cakephp n'est pas le framework le plus moderne, et pourtant, on peut l'adapter pour cet exercice. On conserve ainsi la simplicité d'un framework « rapide », tout en apportant la puissance d'un métier solide. On peut même envisager de conserver ce framework dans le cadre de cette modernisation, en passant à la version 4.

4. ALLER PLUS LOIN DANS LE DDD

Avant d'aller plus loin, regardons de plus près ce que nous avons fait là.

4.1. Value-object

Considérons par exemple la signature de notre bookmarkUpdater

```
namespace Domain\Bookmark\Updater;

class BookmarkUpdater
{
    public function update(
        Bookmark $bookmark,
        string $title,
        string $url,
        string $description,
        array $tagsTitle
    ): ?Bookmark {
        …
    }
}
```

Pas facile de faire confiance à cette variable `$url` pour détenir réellement une adresse web. On ne sait d'ailleurs pas si le protocole est http/s ou un autre. Et si on remplacait cette simple chaîne de caractère en objet métier qui portera le sens de ce qu'elle est !

```
namespace Domain\Bookmark\Updater;

…
use Domain\Bookmark\ValueObject\Url;
```

```
class BookmarkUpdater
{
    public function update(
        …
        string $url,
        Url $url,
    ): ?Bookmark {
        …
    }
}
```

En DDD, on le nomme **value object**. C'est un objet métier, avec toute la valeur que cela implique. Mais à la différence des autres objets métier (comme le bookmark), il n'a pas d'identité propre.

```
namespace Domain\Bookmark\ValueObject;

class Url
{
    public $value;

    public function __construct(string $url) {
        $validation = (bool) preg_match(
            "#(?i)\b((?:https?://|www\
d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+\
)))*\))+(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]{};:'\".,<>?«»“”‘’]))
#",
            $url
        );
        if (!$validation) {
            throw new InvalidValueException('Invalid URL');
        }

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

Je n'ai pas pu résister à y ajouter une validation minimum, avec cette regex trouvée sur stackoverflow que je n'ai même pas testée :P

Au passage, j'ai ajouté cette petite exception tout simple.

```php
namespace Domain\Bookmark\ValueObject;

class InvalidValueException extends \DomainException
{
}
```

Il nous faut aussi réaliser la création du value object quand on récupère les données de la persistance via le repository. On réalise à ce moment que lorsque l'on fait remonter des données depuis la persistance, on passe aussi par la validation. Ce n'est pas pertinent car les règles de validations peuvent changer dans la vie du projet, et on peut se retrouver avec des données stockées invalides. On va donc s'arranger pour créer la value-object quand même.

```php
namespace Domain\Bookmark\ValueObject;

class Url
{
    public $value;

    public static function fromString($url) {
        $validation = (bool) preg_match(
            "#(?i)\b((?:https?://|www\
d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\
)))*\))+(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]{};:'\".,<>?«»“”‘’]))
#",
            $url
        );
        if (!$validation) {
            throw new InvalidValueException('Invalid URL');
        }

        return new self($url);
    }

    public static function fromPersistedString($url) {
        return new self($url);
    }

    private function __construct(string $url) {
        $this->value = $url;
    }
}
```

On pourrait même envisager d'ajouter une validation des données même quand on récupère les données de la base de données. Par exemple, pour éviter un `http://example.com/"><script>alert("Vibby hacked here.")</script>`. Ok, la couche infrastructure est censée éviter ces injections, mais il est bon de savoir qu'on peut faire des vérifications ici.

Il nous faut maintenant ajouter cette logique à la transformation des objet métier Cakephp vers le modèle métier

```
namespace App\Controller\Component;

use Domain\Bookmark\ValueObject\Url;

class BookmarkTransformerComponent extends Component
{
    public function modelToEntity(BookmarkModel $bookmarkModel): Bookmark
    {
        …
        $bookmarkEntity->set('url', $bookmarkModel->url);
        $bookmarkEntity->set('url', $bookmarkModel->url->value);
        …
    }

    public function EntityToModel(Bookmark $bookmarkEntity): BookmarkModel
    {
        …
        $bookmarkModel->url = $bookmarkEntity->get('url');
        $bookmarkModel->url =
Url::fromPersistedString($bookmarkEntity->get('url'));
        …
    }
}
```

Et enfin, nous l'avions oublié, la mise à jour de l'appel à l'updater

```
namespace Application\UpdateBookmark;

use Domain\Bookmark\ValueObject\Url;

class UpdateBookmarkHandler
{
    public function __invoke(
        UpdateBookmarkInput $input
    ): ?Bookmark {
```

```
        $bookmark = $this->bookmarkRepository->findById($input->id);
        try {
            $url = Url::fromString($input->url)
        } catch (InvalidValueException $exception) {
            return null;
        }

        $bookmark = $this->updater->update(
            …
            $input->url,
            $url,
        );
        …
    }
}
```

Je vous conseille vivement de créer les value object le plus tôt possible (dans les handler), et de ne travailler qu'avec ces objets dans toute la partie domaine. Franchement, c'est très confortable de travailler avec des objets que l'on connaît et en qui on peut avoir confiance.

La value-object est tout à fait disposée à recevoir toute validation spécifique à un projet. Par exemple, un value object « pourcentage » pourra valider que sa valeur est bien comprise entre 0 et 100. Ou bien des règles encore plus dirigées par le métier, comme une commission qui ne doit jamais 10%.

Toutes les propriétés peuvent ainsi être adaptées sous la forme de value-object : les chaînes de caractères, les entiers et décimaux, des tableaux… On peut même y stocker des données composées, comme une adresse postale composée d'un numéro, nom de voie, code postal, etc. C'est en fait un choix métier de faire ce regroupement. Et ce sera à la couche infra de se débrouiller pour la stocker, puis de la recréer depuis la persistance.

DDD Une des régles importantes du DDD, c'est que toute classe métier créée doit être valide. Dans tout les cas on doit pouvoir compter sur une base minimale de logique en son sesin. Ceci est valable pour les objets métier (le bookmark) ou les value-object. Cependant, c'est le métier lui-même qui défini ce qu'est un objet valide.

Suivi des modifactions : https://github.com/vibby/cakephp-bookmarker-

tutorial/commit/aff719ccb6cb8e9e473d2411db331630ee99596c

4.2. Validation de l'input

Nous venons de mettre en place un niveau de validation basique au niveau de value-object. Mais il y a d'autres validations qu'il nous faudra mettre en place.

La couche domaine de l'application ne fait confiance à personne. Il nous faut donc mettre en place une validation des demandes qui y sont faites. Voyons donc la validation de input de mise à jour d'un bookmark.

```
namespace Application\UpdateBookmark;

class UpdateBookmarkValidator
{
    public function validate(
        UpdateBookmarkInput $input
    ): array {
        $violations = [];
        if (mb_strlen($input->title) < 3) {
            $violations[] = 'Title must be at last 3 char long';
        }
        if (mb_strlen($input->title) > 1024) {
            $violations[] = 'Title cannot be more than 1024 char long';
        }

        return $violations;
    }
}
```

On placera ici toute sorte de validation qu'on est en mesure de coder. On peut aussi créer des règles de validation basées sur plusieurs champs. Par exemple, la description est obligatoire si le titre fait moins de 10 caractères. Oui, les règles métiers ont parfois leurs raisons que la raison ignore !

Il nous faut maintenant exécuter cette validation en tête du handler.

```
namespace Application\UpdateBookmark;

use Domain\Bookmark\Exception\ViolationCollectionException;
```

```
class UpdateBookmarkHandler
{
    …
    private $validator;

    public function __construct(
        …
        UpdateBookmarkValidator $updateBookmarkValidator
    ) {
        …
        $this->validator = $updateBookmarkValidator;
    }

    public function __invoke(
        UpdateBookmarkInput $input
    ): ?Bookmark {
        $errors = $this->validator->validate($input);
        $bookmark = $this->bookmarkRepository->findById($input->id);
        if (!$bookmark) {
            return null;
            $errors[] = 'Bookmark does not exists.';
        }
        if (count($errors)) {
            throw new ViolationCollectionException('Errors occured with your
request', $errors);
        }
        …

    }
}
```

En plus de lever l'exception en cas d'invalidité de l'input, nous ajoutons aussi le cas où le bookmark est introuvable. Par exemple, si il a été supprimé par un autre utilisateur entre temps.

On a créé ici cette exception qui va se charger de porter la liste des erreurs à la mise à jour, auquels nous avons donné le nom de violation.

```
namespace Domain\Bookmark\Exception;

use Exception;
use Throwable;
```

```
class ViolationCollectionException extends Exception
{
    public $violationCollection;

    public function __construct($message, array $violationCollection, $code =
0, Throwable $previous = null)
    {
        $this->violationCollection = $violationCollection;
        parent::__construct($message, $code, $previous);
    }
}
```

À ce stade, notre application aboutira à une erreur 500 si on essaie de modifier le titre d'un bookmark par « ab » car elle ne respecte pas la règle des 3 caractères. Du point de vue du domaine, la règle métier est respectée, la fonction est remplie.

Il va donc falloir s'adapter côté infrastructure pour en rendre compte correctement à l'utilisateur. Pour cela, nous allons pouvoir intercepter cette exception.

```
namespace App\Controller;

class BookmarksController extends AppController
{
    public function edit($id = null)
    {
        if ($this->request->is(['patch', 'post', 'put'])) {
            …
            try {
                $handler($input);
                $this->Flash->success('The bookmark has been saved.');
                return $this->redirect(['action' => 'index']);
            } catch (ViolationCollectionException $exception) {
                $this->Flash->error(implode(' ',
$exception->violationCollection));
            }
        }
        …
    }
}
```

N'oublions pas d'injecter le validateur via notre container maison.

```
namespace App\Controller\Component;

use Application\UpdateBookmark\UpdateBookmarkValidator;

class ContainerComponent extends Component
{
    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        …
        $this->container[UpdateBookmarkValidator::class] = new
UpdateBookmarkValidator();
        $this->container[UpdateBookmarkHandler::class] = new
UpdateBookmarkHandler(
            $this->BookmarkRepository,
            $this->container[BookmarkUpdater::class],
            $this->container[UpdateBookmarkValidator::class]
        );
    }
}
```

Et nous voici avec de magnifiques messages d'erreur sur la validation qui proviennent directement de la couche domaine. Nous pouvons retirer les validations faites côté Cakephp, dans la classe BookmarkTable.

Le résumé précis dans ce commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/8fa09557b8914d8b55a04b17b37eec888cf943cb

Nous avons mis en place ici la validation sur la base de l'input, mais on pourrait utiliser la même technique pour faire de la validation au niveau de l'objet métier avec des règles différentes. Par exemple, pour ce point d'entrée, on pourrait refuser que l'url soit sur un autre protocole que http. Mais pour l'objet lui-même, il pourrait autoriser aussi ftp ou tout autre protocole.

4.3. Validation basée sur le contexte

Pour aller plus loin, on pourrait avoir des règles de validations basées sur le contexte global. Par exemple, le nom de domaine en cours, ou l'utilisateur

connecté. Commençons par le validateur lui-même.

```
namespace Domain\Bookmark\Validator;

use Domain\Bookmark\Context\CurrentUserProvider;
use Domain\Bookmark\Model\Bookmark;

class BookmarkUpdaterValidator
{
    private $currentUserProvider;

    public function __construct(
        CurrentUserProvider $currentUserProvider
    ) {
        $this->currentUserProvider = $currentUserProvider;
    }

    public function validate(Bookmark $bookmark): array
    {
        $violations = [];
        $currentUser = $this->currentUserProvider->getCurrentUser();
        if ($currentUser === null || $currentUser->id !== $bookmark->user->id)
{
            $violations[] = 'You cannot modify that bookmark since you are not
the owner';
        }
        return $violations;
    }
}
```

Nous utilisons un nouveau service ici pour retrouver l'utilisateur courant.

```
namespace Domain\Bookmark\Context;

use \Domain\Bookmark\Model\User;

interface CurrentUserProvider
{
    public function getCurrentUser(): ?User;
}
```

Il s'agit d'une interface, car nous laissons son implémentation pour l'infrastructure, dans le cadre d'un composant Cakephp.

```
namespace App\Controller\Component;

use App\Model\Entity\User;
use Cake\Controller\Component;
use Cake\Controller\Component\AuthComponent;
use Domain\Bookmark\Context\CurrentUserProvider;
use Domain\Bookmark\Model\User as UserModel;

/**
 * @property UserTransformerComponent $UserTransformer
 * @property AuthComponent $Auth
 */
class CurrentUserProviderComponent extends Component implements
CurrentUserProvider
{
    public $components = ['Auth', 'UserTransformer'];

    public function getCurrentUser(): ?UserModel
    {
        return $this->UserTransformer->EntityToModel(new
User($this->Auth->user()));
    }
}
```

Ajoutons tout de suite ces dépendances à notre container.

```
namespace App\Controller\Component;

use Domain\Bookmark\Validator\BookmarkUpdaterValidator;

/**
* @property CurrentUserProviderComponent $CurrentUserProvider
*/
class ContainerComponent extends Component
{
    public $components = ['BookmarkRepository', 'TagRepository'];
    public $components = ['BookmarkRepository', 'TagRepository',
'CurrentUserProvider'];

    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        …
        $this->container[BookmarkUpdaterValidator::class] = new
```

```
BookmarkUpdaterValidator($this->CurrentUserProvider);
        $this->container[UpdateBookmarkHandler::class] = new
UpdateBookmarkHandler(
            …
            $this->container[BookmarkUpdaterValidator::class]
        );
    }
}
```

Et bien il ne nous reste plus qu'à appliquer ce nouveau validateur au handler.

```
namespace Application\UpdateBookmark;

class UpdateBookmarkHandler
{
    …
    private $updateValidator;

    public function __construct(
        …
        BookmarkUpdaterValidator $bookmarkUpdaterValidator
    ) {
        …
        $this->updateValidator = $bookmarkUpdaterValidator;
    }

    public function __invoke(
        UpdateBookmarkInput $input
    ): ?Bookmark {
        if (!$bookmark) {
            $errors[] = 'Bookmark does not exists.';
        } else {
            $errors = array_merge($errors,
$this->updateValidator->validate($bookmark));
        }
        if (count($errors)) {
            throw new ViolationCollectionException('Errors occured with your
request', $errors);
        }
    }
}
```

Ainsi les erreurs de notre nouveau validateur vont s'ajouter aux autres

erreurs déjà existantes.

Nous sommes maintenant en mesure d'établir des règles de validations basées sur toute sorte d'élucubration que le métier voudra voir implémenter !

Il nous reste une petite chose à faire cependant, c'est de transmettre les erreurs de validations à la création des value-object pour les afficher à l'utilisateur.

```
namespace Application\UpdateBookmark;

class UpdateBookmarkHandler
{
    public function __invoke()
    {
        …
        try {
            $url = Url::fromString($input->url);
        } catch (InvalidValueException $exception) {
            $errors[] = $exception->getMessage();
        }
        if (count($errors)) {
            throw new ViolationCollectionException('Errors occured with your
request', $errors);
        }
        …
        $bookmark = $this->updater->update(
            Url::fromString($input->url),
            $url,
        )
        …
    }
}
```

Le résumé en code ici : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/ee34d69633084c34c1094217f6a37558502a2e57

Maintenant que nous avons un peu de code dans le domaine, il est clair que ce cœur de l'application est la partie la plus cruciale de l'application. La tester untiairement est donc sûrement une très bonne idée. Gardons cette tâche pour le chapitre 5.6, quand nous aurons une gestion des dépendances plus complète.

4.4. Langage ubiquiste

Dans le projet sur lequel nous travaillons ici, les termes utilisés sont **Bookmark**, **Url**, **Tag**, etc. Mais il pourrait survenir que ces termes ont été choisis par le développeur au moment de réaliser concrètement les implémentations.

Mais si on parle avec d'autres acteurs du projet, on va peut-être réaliser qu'il a un décalage de vocabulaire entre les acteurs. Ce décalage est source d'incompréhensions et donc d'erreurs. D'ailleurs, ce n'est pas les idées qui forment les mots, mais bien les mots qui forment les idées.

C'est pourquoi le DDD nous propose d'utiliser un langage ubiquiste, c'est-à-dire universel tous les acteurs du projet. Ça peut donner parfois des résultats étranges, notamment quand le métier parle en français. On se retrouve avec du bon franglais dans le code, comme une méthode *getSociete* ou bien une classe *CreerSocieteHandler*. Mais le gain en efficacité du projet en réel !

DDD Le langage ubiquiste, unviversel pour le projet, participe à éviter de créer le décalage entre ce que disent les acteurs du projet, et ce que l'on peut lire dans le code.

5. ASSURER L'AVENIR

5.1. Montée de version

Il est temps de passer à la mise à jour de toutes ces vielles sources. On va passer à la dernière version de Cakephp en deux étapes.

```
docker-compose exec web composer require --update-with-dependencies "cakephp/
cakephp:3.10.*"
```

On se retrouve avec un bon paquet de dépréciations relativement facile à traiter. Je vous passe les détails que vour pourrez consulter dans ce commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/ee69151acdce62c80da599580bd4ac883c0c6082

Passons à php 7.4 et composer 3 pour installer cakephp 4 dans de bonnes conditions.

```
# .docker/php/Dockerfile
FROM php:7.1-fpm-buster
FROM php:7.4-fpm-buster
…
COPY --from=composer:1 /usr/bin/composer /usr/local/bin/composer
COPY --from=composer /usr/bin/composer /usr/local/bin/composer
…
```

```
docker-composer stop
docker-composer up --build
```

Pour mettre à jour notre base de code, nous allons utiliser l'outil (rector)[https://getrector.org/], c'est un outil d'automatisation de la modernisation du code lorsqu'il n'y a pas d'ambiguïté. Par exemple, si une classe implémente une interface, il va mettre à jour les signatures des méthodes de la classe, identiques à celles de l'interface.

D'ailleurs, Cakephp nous propose un outil pour automatiser cette montée

de version, qui se base sur Rector.

```
docker-compose exec web bash
git clone https://github.com/cakephp/upgrade
cd upgrade
git checkout 4.x
composer install --no-dev
bin/cake upgrade filename templates /app/src/cakephp/Template/
bin/cake upgrade rector --rules phpunit80 /app/tests/
bin/cake upgrade rector --rules cakephp40 /app/src/cakephp/
bin/cake upgrade rector --rules cakephp41 /app/src/cakephp/
bin/cake upgrade rector --rules cakephp42 /app/src/cakephp/
bin/cake upgrade rector --rules cakephp43 /app/src/cakephp/
bin/cake upgrade rector --rules cakephp44 /app/src/cakephp/
exit
```

On peut maintenant supprimer l'outil de montée de version

```
docker-compose exec web rm -rf upgrade
```

Nous voilà prêt pour le passage sur cakephp 4.4 à proprement parler, la dernière en date.

```
docker-compose exec web composer require --dev --update-with-dependencies
"phpunit/phpunit:^8.0"
docker-compose exec web composer remove cakephp/migrations cakephp/bake
cakephp/debug_kit --dev
docker-compose exec web composer require --update-with-dependencies "cakephp/
cakephp:4.4.*"
docker-compose exec web composer require cakephp/migrations:* cakephp/bake:*
cakephp/debug_kit --dev --update-with-all-dependencies
```

Et puisque nous sommes si bien lancés, passons aussi à la dernière version de PHP.

```
# .docker/php/Dockerfile
~~FROM php:7.4-fpm-buster~~
FROM php:8.1-fpm-buster
…
```

```
docker-composer stop
docker-composer up --build
```

On remplit à nouveau la base de données

```
docker-compose exec db bash
mysql -u root -pbookmark < /dumps/app.sql bookmark
mysql -u root -pbookmark < /dumps/i18n.sql bookmark
mysql -u root -pbookmark < /dumps/sessions.sql bookmark
exit
`
```

On trouve alors des erreurs liées au fort typage introduit dans cakephp 4. Malgré ses efforts, Rector ne les a pas toutes réglées. Pour y palier, il nous suffit de mettre à jour les fichiers de cakephp qui sont versionnés mais qui ont évolués, à partir de https://github.com/cakephp/app/tree/4.x/src. C'est le cas notamment des fichiers `bin/cake`, `config/bootstrap`, `src/cakephp/Application.php`, etc.

Autre chose : les fichiers des template changent d'extension et passent de .ctp à .php ce qui me semble bien plus naturel puisque ce sont précisément des fichiers php. Et cakephp propose de les placer plutôt à la racine du projet. Il y a encore quelques petits ajustements, je vous laisse le loisir de voir cela dans ce commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/43519dd08804ec6637e0b52ef9c58a6ad0aebc38

Finalement, on revient rapidement à notre application fonctionnelle !

> **DDD** On récolte ici le fruits de nos efforts ! La séparation des couches de l'application nous a permis une montée de version de l'infrastructure simplifiée.

Si vous en avez déjà fait, vous savez que c'est un travail long et fastidieux. En minimisant la couverture de responsabilité de la couche infra, on simplifie grandement son évolution et sa maintenance. On touche du doigt un des énormes intérêts du DDD. D'ailleurs, dans l'interface de l'application, on ne rencontre pas de dysfonctionnement pour les fonctionnalités que nous avons passées en couche infra + application + domaine. Mais le reste des fonctionnalités rencontre des exceptions. Par

exemple, la création d'utilisateur qui aboutit à l'erreur `too few arguments to function Cake\ORM\Table::newEntity()`.

5.2. Dépendances et php-arkitect

> **DDD** Nous l'avons déjà précisé, les dépendances entre les couches de l'application ne se font que dans un seul sens : `[infrastructure] -> [Application] -> [Domaine]`

L'intérêt de cette contrainte, c'est que le domaine conserve une bonne indépendance de l'infrastructure. Et en cas de modification au niveau de l'infrastructure, elle ne devrait jamais impacter le domaine. En voici quelques exemples :

- mise à jour du framework (comme on vient de le voir)
- mettre en place un moteur de recherche pour indexer les données
- utilisation d'un système de messaging en asynchrone pour l'envoi d'email

Par contre, une modification au niveau de la couche domaine peut tout à fait avoir des répercussions au niveau de l'infrastructure. Comme par exemple l'ajout d'un paramètres à un entrypoint.

Pour s'assurer que cette règle restera respectée dans le temps, nous allons mettre en place un outil d'analyse : (phpaskitect)[https://github.com/phparkitect/arkitect].

```
docker-compose exec web composer require phparkitect/phparkitect --dev
```

Écrivons maintenant la règle toute simple que la couche domaine ne doit dépendre d'aucun autre domaine de nom.

```
// ./phparkitect.php
declare(strict_types=1);

use Arkitect\ClassSet;
use Arkitect\CLI\Config;
use Arkitect\Expression\ForClasses\NotHaveDependencyOutsideNamespace;
```

```
use Arkitect\Expression\ForClasses\ResideInOneOfTheseNamespaces;
use Arkitect\Rules\Rule;

return static function (Config $config): void {
    $mvcClassSet = ClassSet::fromDir(__DIR__.'/src');

    $rules = [];

    $rules[] = Rule::allClasses()
        ->that(new ResideInOneOfTheseNamespaces('Domain'))
        ->should(new NotHaveDependencyOutsideNamespace('Domain'))
        ->because('we want the domain independent from the outside');

    $config
        ->add($mvcClassSet, ...$rules);
};
```

On lance phpakitect avec la commande suivante

```
docker-compose exec web vendor/bin/phparkitect check
```

C'est un échec, phparkitect nous a trouvé deux violations.

```
Domain\Bookmark\Exception\ViolationCollectionException has 2 violations
  depends on Exception, but should not depend on classes outside namespace
Domain because we want a domain independent from the outside (on line 8)
  depends on Throwable, but should not depend on classes outside namespace
Domain because we want a domain independent from the outside (on line 12)
```

Puisque nous avons défini que nous ne voulions aucune dépendance, il a identifié la classe `\Exception` et l'interface `\Throwable` comme des dépendances extérieures. Nous allons donc les inclure parmi les classes autorisées.

```
// ./phparkitect.php
    …
    $rules[] = Rule::allClasses()
        ->that(new ResideInOneOfTheseNamespaces('Domain'))
        ~~->should(new NotHaveDependencyOutsideNamespace('Domain'))~~
        ->should(new NotHaveDependencyOutsideNamespace('Domain', ['Exception',
```

```
'Throwable',]))
        ->because('we want the domain independent from the outside');
        …
```

On a le même souci avec la dépendance à `LogicException` depuis la couche application.

Cette identification a la vertu de garder une vision claire et nette des dépendances de la couche domaine.

Vérifions aussi les dépendances de la couche application.

```
// ./phparkitect.php
    …
    $rules[] = Rule::allClasses()
        ->that(new ResideInOneOfTheseNamespaces('Application'))
        ->should(new NotHaveDependencyOutsideNamespace('Application',
['Domain',]))
        ->because('we want the application layer dependent on domain only');
        …
```

Tant que nous sommes dans les règles, laissez moi vous montrez comment phparkitect peut aussi nous aider un code propre côté framework avec les règles suivantes.

```
// ./phparkitect.php
…
    $rules[] = Rule::allClasses()
        ->that(new ResideInOneOfTheseNamespaces('App\Controller\Component'))
        ->should(new HaveNameMatching('*Component'))
        ->because('we want uniform naming for components');

    $rules[] = Rule::allClasses()
        ->that(new ResideInOneOfTheseNamespaces('App\Controller'))
        ->andThat(new NotResideInTheseNamespaces('App\Controller\Component'))
        ->should(new HaveNameMatching('*Controller'))
        ->because('we want uniform naming for controllers');
…
```

La syntaxe de phparkitect parle d'elle même que j'ai à peine besoin de paraphraser : *Toute classe dans le nom de domaine* `App\Controller\Component` *devrait avoir un nom qui ressemble à* `Component` pour garder

une cohérence dans le nommage des composants.*

Nouveau commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/5a522f3efee6b04356044af413052737d3ef1ea1

5.3. php-cs-fixer

Php-cs-fixer est un outil de formatage du code, en respect des standards et des normes, notamment les PSR-1 et 2. Il s'installe dans un dossier à part afin d'éviter des conflits inutiles entre dépendances, puisque c'est un outil indépendant.

```
docker-compose exec web mkdir --parents tools/php-cs-fixer
docker-compose exec web composer require --working-dir=tools/php-cs-fixer
friendsofphp/php-cs-fixer
```

Puis on lance la correction automatisée

```
docker-compose exec web tools/php-cs-fixer/vendor/bin/php-cs-fixer fix src
```

Il nous a trouvé une bonne vingtaine de corrections à réaliser. En voici un exemple :

```
--- a/src/Domain/Bookmark/ValueObject/Url.php
+++ b/src/Domain/Bookmark/ValueObject/Url.php
@@ -6,7 +6,8 @@ class Url
 {
     public $value;

-    public static function fromString($url) {
+    public static function fromString($url)
+    {
```

Il ne réalise aucune modification qui puisse présenter un risque pour le fonctionnement de l'application. Les règles que php-cs-fixer applique sont discutables sur le plan de l'efficacité, mais elles ont l'énorme avantage d'uniformiser tout code. On se sent toujours plus à l'aise dans un format que l'on retrouve ! On va le configurer pour lui demander d'aller un peu plus loin, en utilisant toutes les règles sous le nom «PhpCsFixer».

```
// ./.php-cs-fixer.php
$finder = PhpCsFixer\Finder::create()
    ->in(__DIR__ . '/src');

$config = new PhpCsFixer\Config();

return $config->setRules([
    '@PhpCsFixer' => true,
    'array_syntax' => ['syntax' => 'short'],
])->setFinder($finder);
```

Puis on relance la correction automatisée sans préciser l'emplacement des sources puisque nous l'avons défini dans le fichier de configuration.

```
docker-compose exec web tools/php-cs-fixer/vendor/bin/php-cs-fixer fix
```

Il nous a trouvé à nouveau 25 corrections. On envoi tout ça dans un nouveau commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/3584c595b9472d2b557e011ba716b317af947175

5.4. phpstan

Ajoutons maintenant un autre outil d'analyse statique qui tâche d'identifier de mauvaises pratiques au sein de l'application.

```
 docker-compose exec web composer require --dev phpstan/phpstan
```

Et lançons le.

```
docker-compose exec web vendor/bin/phpstan analyse src
```

Une seule erreur est identifiée.

```
 ------
--------------------------------------------------------------------------------
---
  Line   cakephp/Controller/
BookmarksController.php
 ------
```

```
------------------------------------------------------------------------------
---
  173    Call to an undefined static method
App\Controller\AppController::isAuthorized().
 ------
------------------------------------------------------------------------------
---
```

Et effet, c'est un reliquat de la façon de fonctionner du système d'autorisation de cakephp. On va dire que les utilisateurs ne sont pas autorisés par défaut pour le moment.

Phpstan offre 10 niveaux de vérification, de 0 (par défaut) à 9. Laissons de côté le code lié à Cakephp et lançons l'analyse sur les parties domaine et application uniquement.

```
docker-compose exec web vendor/bin/phpstan analyse -l 9 ./src/Application
./src/Domain
…


 [ERROR] Found 40
errors
```

Joie ! Voilà de quoi améliorer notre code !

Prenons une erreur en exemple.

```
 ------
------------------------------------------------------------------------------
---------------------------
  Line   Bookmark/ValueObject/
Url.php

 ------
------------------------------------------------------------------------------
---------------------------
  7      Property Domain\Bookmark\ValueObject\Url::$value has no type
specified.
```

```
 ------
--------------------------------------------------------------------------------
-------------------------
```

L'erreur est limpide, la propriété `value` de l'objet `Url` n'est pas typée. Et pour cause, ce n'était pas possible dans notre php 7.1 original. Les erreurs rencontrées sont en fait de 4 types :

- Pas de typage pour une propriété
- Pas de typage pour un paramètre de méthode
- Pas de typage de retour de fonction
- Pas de typage pour les éléments d'un tableau

En fait, phpstan va plus loin que les contrôles de php pour le typage et donc la cohérence globale de l'application. Et c'est bien ! Traitons tous ces points.

Si on lance l'analyse sur la partie infra maintenant.

```
docker-compose exec web vendor/bin/phpstan analyse -l 9 ./src/cakephp
…


 [ERROR] Found 93
errors
```

Il y a là bien plus de travail, nous n'allons pas les traiter. Mais on constate avec bonheur que les parties que nous avons passées sous le paradigme du DDD ne sont que très faiblement touchées ! Encore une bonne raison d'avoir sauté le pas.

Je vous épargne les détails des corrections, mais vous les trouverez dans le commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/f8788dc6f3b95389fd5a1c5de9a8201c69092268

5.5. Évolutions de PHP

À chaque nouvelle version, PHP apporte ces améliorations et nous offre de nouvelles fonctionnalités. Puisque nous en sommes à moderniser, utilisons

les dernières évolutions de PHP. En voici trois exemples.

Définir des propriétés directement depuis le constructeur d'une classe.

```
namespace Application\GetBookmark;

class GetBookmarkHandler
{
    private BookmarkRepository $bookmarkRepository;

    public function __construct(
        BookmarkRepository $bookmarkRepository
        private BookmarkRepository $bookmarkRepository
    ) {
        $this->bookmarkRepository = $bookmarkRepository;
    }
    …
}
```

Définir des propriétés en readonly.

```
namespace Application\GetBookmark;

class GetBookmarkInput
{
    public int $id;
    public readonly int $id;
}
```

Nommer les paramètres à l'appel d'une fonction.

```
namespace App\Controller;

class BookmarksController extends AppController
{
    public function view($id = null)
    {
        $input = new GetBookmarkInput($id);
        $input = new GetBookmarkInput(id: $id);
        …
    }
}
```

Une fois que tout cela est réalisé, mettons à profit nos outils de bonne tenue de code.

```
docker-compose exec web tools/php-cs-fixer/vendor/bin/php-cs-fixer fix
docker-compose exec web vendor/bin/phparkitect check
docker-compose exec web vendor/bin/phpstan analyse -l 9 ./src/Application
./src/Domain
```

Quand tout est vert, encapsulons ces modifications dans un nouveau commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/d971a2f0d10865e008363f2b37249dc12a4b8758

DDD La couche domaine doit aussi évoluer avec ces dépendances, même si nous l'avons limité autant que possible. Mais elle dépend bien entendu de PHP, et peut-être quelques classes natives du langage, comme `\DateTime`.

6. TESTER

6.1. un cœur éprouvé par phpunit

Revenons dans la continuité du point 4.3. Passons à l'écriture de quelques test phpunit. Mais avant cela, il va nous falloir faire de la place car Cakephp vient avec sa propre configuration de PHPunit.

```
mv tests Cakephp
mkdir tests
mv Cakephp tests
mv phpunit.xml.dist tests/Cakephp/phpunit.xml.dist
```

Après cela il faut changer quelques chemins dans `tests/Cakephp/phpunit.xml` et `tests/Cakephp/bootstrap.php`, puis l'ensemble des noms de domaine des tests cakephp.

```
    ~~namespace App\Test\TestCase\Controller;~~
    namespace App\Test\Cakephp\TestCase\Controller;
```

Les tests cakephp peuvent ainsi toujours être lancés avec cette commande.

```
docker-compose exec web vendor/bin/phpunit --config tests/Cakephp/
phpunit.xml.dist
```

À noter qu'ils ne passent pas dans l'actuel car il ne trouve pas les fixtures. Mais nous pouvons maintenant écrire nos propres tests. Savoir identifier quelles parties doivent être testées unitairement en priorité est simplifié par la découpe du code. Notamment, les validateurs que nous avons définis méritent toute notre attention pour pouvoir leur faire confiance. Écrivons donc un test en exemple pour le validateur de l'input de modification d'un bookmark.

```
namespace App\Test\Application\UpdateBookmark;

use Application\UpdateBookmark\UpdateBookmarkInput;
use Application\UpdateBookmark\UpdateBookmarkValidator;
```

```php
use PHPUnit\Framework\TestCase;

class UpdateBookmarkValidatorTest extends TestCase
{
    private UpdateBookmarkValidator $validator;

    public function setUp(): void
    {
        $this->validator = new UpdateBookmarkValidator();
    }

    public function testValidInput()
    {
        $input = new UpdateBookmarkInput(
            id: 12,
            title: 'Just a title',
            url: 'http://example.com',
            description: 'A simple descr',
            tagsTitle: [],
        );

        $this->assertEmpty($this->validator->validate($input));
    }

    public function testTitleTooShort()
    {
        $input = new UpdateBookmarkInput(
            id: 12,
            title: 'Ju',
            url: 'http://example.com',
            description: 'A simple descr',
            tagsTitle: [],
        );

        $this->assertEquals($this->validator->validate($input), ['Title must
be at last 3 char long.']);
    }
}
```

On met ici en évidence que notre code est découpé de manière très atomique : chaque partie ne fait qu'une petite chose. Et ça devient alors très facile de la tester et de gagner en confiance sur le comportement de l'application. Le gain n'est pas évident dans ce cas, mais sur un projet complexe, on aura d'autant plus éclaté la complexité en petites briques parfaitement appréhendables.

Pour lancer ce test, il va nous falloir configurer phpunit dans le fichier `./phpunit.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/9.3/phpunit.xsd"
    colors="true"
>
    <testsuites>
        <testsuite name="Application">
            <directory>tests/Application/</directory>
        </testsuite>
        <testsuite name="Domain">
            <directory>tests/Domain/</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Puis lancer les tests.

```
docker-compose exec web vendor/bin/phpunit
```

Pour compléter cette partie sur les tests, voici le tests du validateur d'update d'un bookmark

```
namespace App\Test\Domain\Bookmark\Validator;

use Domain\Bookmark\Context\CurrentUserProvider;
use Domain\Bookmark\Model\Bookmark;
use Domain\Bookmark\Model\User;
use Domain\Bookmark\Validator\BookmarkUpdaterValidator;
use PHPUnit\Framework\TestCase;

class BookmarkUpdaterValidatorTest extends TestCase
{
    private BookmarkUpdaterValidator $validator;

    public function setUp(): void
    {
        $this->currentUserProvider =
$this->createMock(CurrentUserProvider::class);
        $this->validator = new
BookmarkUpdaterValidator($this->currentUserProvider);
```

```
    }

    public function testSameUser()
    {
        $user = $this->createMock(User::class);
        $user->id = 12;

$this->currentUserProvider->method('getCurrentUser')->willReturn($user);

        $bookmark = $this->createMock(Bookmark::class);
        $user2 = $this->createMock(User::class);
        $user2->id = 12;
        $bookmark->user = $user2;

        self::assertEmpty($this->validator->validate($bookmark));
    }

    public function testDifferentUser()
    {
        $user = $this->createMock(User::class);
        $user->id = 12;

$this->currentUserProvider->method('getCurrentUser')->willReturn($user);

        $bookmark = $this->createMock(Bookmark::class);
        $user2 = $this->createMock(User::class);
        $user2->id = 13;
        $bookmark->user = $user2;

        self::assertEquals($this->validator->validate($bookmark), ['You cannot
modify that bookmark since you are not the owner']);
    }

    public function testCannotFindCurrentUser()
    {
        $this->currentUserProvider->method('getCurrentUser')->willReturn(null);

        $bookmark = $this->createMock(Bookmark::class);
        $user2 = $this->createMock(User::class);
        $user2->id = 13;
        $bookmark->user = $user2;

        self::assertEquals($this->validator->validate($bookmark), ['You cannot
modify that bookmark since you are not the owner']);
    }
}
```

Comme notre validateur a besoin du système pour retrouver l'utilisateur courant, on vient le simuler par un *mock*. Ensuite on lui fait faire ce qu'on veut selon les cas que l'on suite produire.

Puis lancer les tests.

```
docker-compose exec web vendor/bin/phpunit
```

```
Time: 24 ms, Memory: 6.00 MB

OK (5 tests, 5 assertions)
```

Voilà une bonne petite stack de tests, prête à recevoir tout les tests unitaires que vous voudriez écrire ! Pour clore cette partie, nous pouvons aussi appliquer php-cs-fixer aux tests depuis le fichier de configuration `.php-cs-fixer.php`

```
$finder = PhpCsFixer\Finder::create()
    ->in(__DIR__ . '/src')
    ->in(__DIR__ . '/tests');
    …
```

```
docker-compose exec web tools/php-cs-fixer/vendor/bin/php-cs-fixer fix src
```

Je compile tout ça dans ce commit pour vous : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/9c04e75c5646ead8ed2eab5482eaa5696e591cdc

6.2. Tests de bout en bout

Les tests unitaires sont pertinents pour s'assurer que les différents éléments rempliront bien leur rôle. Cependant, on ne peut pas se passer des tests d'un utilisateur humain qui va réellement se connecter à l'application et réaliser les différentes actions pour s'assurer que tout est opérationnel.

Alors, on va s'équiper d'un outil qui peut faire ce boulot de manière automatisée ! Il y a quelques années, j'aurais proposé Behat pour cette tâche, mais aujourd'hui, voici (Cypress)[https://www.cypress.io/] ! Ajoutons-le à notre projet. Pour cela vous aurez besoin d'avoir pré-installé nodejs 14+ et npm.

```
npm init
npm install cypress --save-dev
```

On peut lancer immédiatement l'interface de cypress

```
npx cypress open
```

Écrivons notre premier test sans attendre.

```
// cypress/e2e/bookmark.cy.js
describe('Bookmarks management', () => {
    it('shows bookmarks', () => {
        cy.visit('http://0.0.0.0:9050');
        cy.get('input[name=email]').type('user@example.com');
        cy.get('input[name=password]').type('password');
        cy.get('button').click();

        cy.url().should('contain', '/bookmarks')
        cy.get('table tbody tr').should('have.length', 5);
        cy.contains('LetsEncrypt').parent('tr').contains('View').click();

        cy.url().should('contain', '/bookmarks/view/')
        cy.contains('https://letsencrypt.org');
        cy.contains('Free open Certificate Authority');
    })
})
```

Ce test réalise les actions suivantes :

- Chargement de la page de login
- On entre email, mot de passe, puis on valide la connexion
- On vérifie qu'on est bien sur la page de la liste des bookmarks, puis on clic sur l'élément *Let's encrypt*
- On vérifie qu'on est bien sur la page pour afficher un bookmark et son contenu

Cypress nous offre même un affichage en temps réel des actions qu'il réalise, et il s'arrête quand une condition n'est pas remplie.

Voilà un bien faible effort pour apporter une grosse valeur au projet : on sait que les fonctions essentielles sont remplies !

On voit cependant que ces tests sont dépendants de l'état courant de l'application, ce qui peut mener à des erreurs. Par exemple, si on renomme le bookmark *LetsEncrypt* en *Free SSL Service*, on voit bien que le test ne fonctionnera plus puisqu'il ne trouvera pas la chaîne de caractères. Pour résoudre ce problème, nous allons mettre en place un reset des données que nous réalisons avant chaque test, pour être sûr de nos données.

```
// cypress/e2e/bookmark.cy.js
describe('Bookmarks management', () => {
    before(() => {
        cy.request('http://0.0.0.0:9050/dataReset/dbReset');
    });
    …
})
```

Puis il nous implémenter ce contrôleur côté Cakephp

```
namespace App\Controller;

use Cake\Http\Response;
use Cake\Datasource\ConnectionManager;

class DataResetController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->Auth->allow('dbReset'); // Pour dire à Cakephp que les anonymes
peuvent accéder à ce contrôleur
    }

    public function dbReset()
    {
        $conn = ConnectionManager::get('default');
        $conn->execute(file_get_contents(__DIR__.'/../../../config/schema/
app.sql'));
        $conn->execute(file_get_contents(__DIR__.'/../../../config/schema/
i18n.sql'));
        $conn->execute(file_get_contents(__DIR__.'/../../../config/schema/
sessions.sql'));

        return new Response();
    }
```

```
    public function isAuthorized()
    {
        return true;
    }
}
```

Nous l'avons fait ici d'une manière rapide et non sécurisée. Pour une véritable application, il aurait été bien mieux de l'autoriser uniquement en environnement de test et de dev, et d'utiliser le système de fixtures de Cakephp. On va s'en tenir au minimum fonctionnel. Ensuite, on voit qu'on peut immédiatement mettre en commun certaines parties du code du test, à commencer par le nom de domaine pour atteindre l'application. Un peu de configuration ici : `./cypress.config.js`

```
const { defineConfig } = require("cypress");

module.exports = defineConfig({
    e2e: {
        baseUrl: 'http://0.0.0.0:9050',
    },
});
```

```
// cypress/e2e/view-bookmark.cy.js
describe('Bookmarks management', () => {
    before(() => {
        ~~cy.request('http://0.0.0.0:9050/dataReset/dbReset');~~
        cy.request('/dataReset/dbReset');
    });

    it('shows bookmarks', () => {
        ~~cy.visit('http://0.0.0.0:9050');~~
        cy.visit('/');
        …
    })
})
```

Autre point : on commence le test par se connecter à l'application. On se doute bien que ce sera une action réalisée de manière récurrente, on va donc la placer à part.

```
// cypress/support/commands.js
Cypress.Commands.add(
    "userConnect",
```

```
    (email = 'user@example.com', password = 'password') => {
        cy.visit('/users/login');
        cy.get('input[name=email]').type(email);
        cy.get('input[name=password]').type(password);
        cy.get('button').click();
        cy.url().should('contain', '/bookmarks')
    }
);
```

Puis on s'en sert dans notre test

```
describe('Bookmarks management', () => {
    it('shows bookmarks', () => {
        cy.visit('/');
        cy.get('input[name=email]').type('user@example.com');
        cy.get('input[name=password]').type('password');
        cy.get('button').click();
        cy.userConnect();

        cy.url().should('contain', '/bookmarks');
        cy.get('table tbody tr').should('have.length', 5);
        cy.contains('LetsEncrypt').parent('tr').contains('View').click();

        cy.url().should('contain', '/bookmarks/view/')
        cy.contains('https://letsencrypt.org');
        cy.contains('Free open Certificate Authority');
    })
})
```

Ajoutons un nouveau test pour modifier un bookmark

```
describe('Bookmarks management', () => {
    …
    it('modifies a bookmark', () => {
        cy.userConnect();

        cy.url().should('contain', '/bookmarks')
        cy.contains('LetsEncrypt').parent('tr').contains('Edit').click();

        cy.url().should('contain', '/bookmarks/edit/')
        cy.get('input[name="title"]').type(" goood");
        cy.get('textarea[name="description"]').type(" Some more content.");
        cy.get('button[type="submit"]').click();
```

```
        cy.url().should('contain', '/bookmarks')
        cy.contains('The bookmark has been saved.');
        cy.contains('LetsEncrypt goood');
    })
})
```

Nous voici avec un joli petit outil, qui va nous assurer que les fonctions essentielles de l'application sont toujours remplies !

À noter que Cypress peut être lancé sans présentation visuelle, très utile pour lancer les tests dans le cadre d'une plateforme d'intégration continue. Il génère même des vidéos du déroulé des actions, précieux pour identifier les raisons d'une erreur.

```
npx cypress run
```

Le résumé dans ce commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/f7a71e3f6a4d81e2c76c610c870c01fb139f7ae6

6.3. Une pseudo CI

Pour résumer, on s'est fait une belle petite stack de vérification de la qualité du code produit !

Si on osait, on exécuterai ces vérifications au moment de créer un commit. On va éditer le fichier `./git/hooks/pre-commit`

```
#!/bin/sh

EXIT_STATUS=0
docker-compose exec -T web tools/php-cs-fixer/vendor/bin/php-cs-fixer fix  ||
EXIT_STATUS=$?
docker-compose exec -T web vendor/bin/phpstan analyse -l 9 ./src/Application
./src/Domain || EXIT_STATUS=$?
docker-compose exec -T web vendor/bin/phpunit || EXIT_STATUS=$?
docker-compose exec -T web vendor/bin/phparkitect check || EXIT_STATUS=$?
exit $EXIT_STATUS
```

Cette technique n'est pas toujours pertinente, car on a parfois juste envie

de créer un commit rapidement. Mais il est aussi très pratique de connaître au plus tôt les erreurs provoqués par nos derniers changements sans avoir à attendre une CI souvent surchargée. Quoi qu'il en soit, je vous conseille de limiter les vérifications faites ici aux plus rapides, c'est-à-dire d'en exclure les tests de bout en bout de cypress.

7. FAIRE DU NEUF

7.1. Nouvelle infra en cohabitation

À présent que nous avons une situation bien saine et solide, le métier vient vers nous et nous annonce que Google est intéressé par notre projet et souhaite communiquer avec notre application, via son API. Bien entendu, nous n'avons rien mis en place de ce côté.

C'est l'occasion rêvée pour relever le défi et mettre en place l'outil d'api parfait : (Api-Platform)[https://api-platform.com/].

Avant de commencer, nous allons isoler la configuration de Cakephp pour ne pas entrer en conflit avec celle de Symfony.

```
mv config configCakephp
```

Il nous faut alors configurer cakephp dans ce sens

```
// configCakephp/paths.php
    …
    define('CONFIG', ROOT . DS . 'config' . DS);
    define('CONFIG', ROOT . DS . 'configCakephp' . DS);
    …
```

```
// webroot/index.php
    …
    $server = new Server(new Application(dirname(__DIR__) . '/config'));
    $server = new Server(new Application(dirname(__DIR__) . '/configCakephp'));
    …
```

Continuons en installant les briques essentielles de Symfony, puis api-plaform.

```
docker-compose exec web composer require symfony/framework-bundle:*
--with-all-dependencies
docker-compose exec web composer require symfony/yaml symfony/runtime symfony/
flex symfony/dotenv api-platform/api-pack
docker-compose exec web composer require --dev symfony/profiler-pack
```

Il nous faut maintenant diriger les requêtes HTTP vers Cakephp ou Symfony selon le path demandé.

```
# .docker/php/vhost.conf
upstream phpfcgi {
    server localhost:9000;
}

server {
    listen 80;
    server_name localhost;
    root /app/public;
    client_max_body_size 80M;

    location / {
        try_files $uri @rewriteapp;
    }

    location /api {
        rewrite ^(.*)$ /symfony.php/$1 last;
    }

    location /_profiler {
        rewrite ^(.*)$ /symfony.php/$1 last;
    }

    location @rewriteapp {
        rewrite ^(.*)$ /index.php/$1 last;
        rewrite ^(.*)$ /cakephp.php/$1 last;
    }

    location ~ ^/index.php(/|$) {
    location ~ ^/.*.php(/|$) {
        include fastcgi_params;
        fastcgi_pass localhost:9000;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

```
        fastcgi_param HTTPS off;
    }
}
```

il nous faut rassembler les deux dossier racine pour le serveur http.

```
mv public/index.php webroot/symfony.php
rm public
mv webroot public
mv public/index.php public/cakephp.php
```

On relance pour prendre en compte ces changement

```
docker-compose up --build
```

Puis, comme nous l'avons fait pour Cakephp, nous allons limiter symfony à une partie du dossier src

```
mkdir src/Symfony
mv src/Controller src/Symfony/Controller
rm src/Repository
rm src/Entity
rm src/ApiResource
mv src/Kernel.php src/Symfony/Kernel.php
```

Ajoutons cette nouvelle partie des noms de domaine à composer.json

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

Ne pas oublier de mettre à jour l'autoload

```
docker-compose exec web composer dump-autoload
```

On rencontre un autre souci également. Avec nos noms de domaines

non standard dans le dossier src, on commence à avoir quelques soucis désagréables. Nous allons donc les uniformiser en les commençant tous pas `App/`.

```
namespace Application\GetBookmark;
namespace App\Application\GetBookmark;

use Domain\Bookmark\Model\Bookmark;
use App\Domain\Bookmark\Model\Bookmark;
use Domain\Bookmark\Repository\BookmarkRepository;
use App\Domain\Bookmark\Repository\BookmarkRepository;
…
```

Puis mettre à jour le composer.json en fonction de cela.

```
    …
    "autoload": {
        "psr-4": {
            "App\\Application\\": "src/Application",
            "App\\Domain\\": "src/Domain",
            "App\\Symfony\\": "src/Symfony",
            "App\\": "src/cakephp"
        }
    },
    …
```

```
docker-compose exec web composer dump-autoload
```

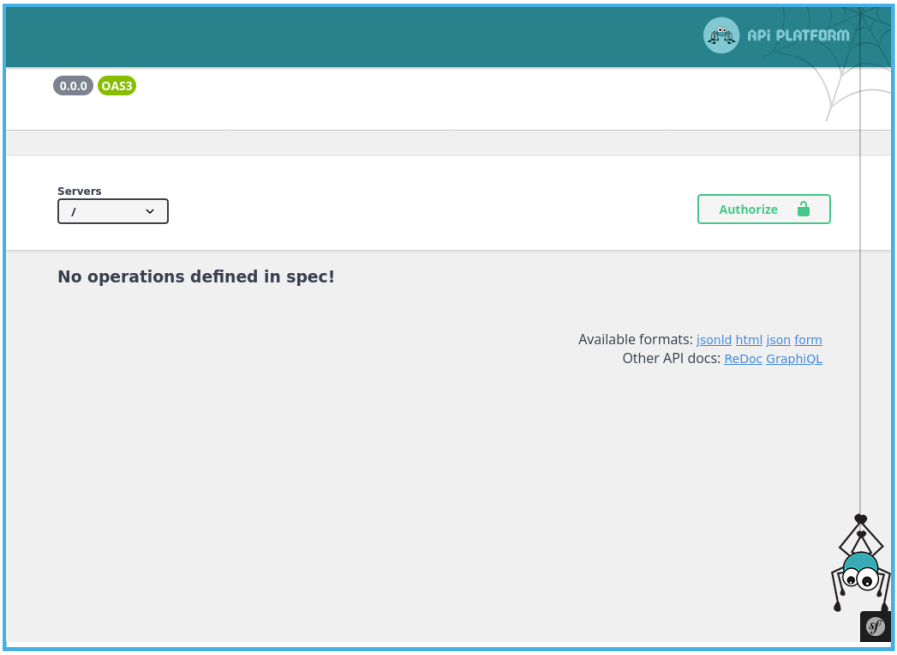
Je vous épargne toutes les modifications mais vous les trouverez dans le commit que voici : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/270537eff57ce397208158681e38d93b4864b28d

Il nous faut également demander à Symfony de publier les assets de ses bundles pour les rendre accessibles dans le dossier public.

```
docker-compose exec web bin/console assets:install --symlink
`
```

Au final, ça fonctionne ! Quand on se rend sur http://0.0.0.0:9050/users/login, on peut se connecter et réaliser toutes les opérations dans l'application Cakephp. Et quand on se rend sur http://0.0.0.0:9050/api/docs,

on voit la page de swagger de description de l'Api, qui est vide pour le moment.



7.2. Nouvelle Doctrine

Nous allons utiliser Doctrine, l'ORM qui poussé par Symfony. Mais Doctrine a un paradigme opposé à l'ORM de Cakephp. La référence du modèle de données est basé sur les fichiers de définition des entités pour Doctrine, alors que Cakephp se base sur la structure de la base de données déjà existante.

Pour être en mesure d'utiliser là même base de données pour les deux infrastructures, il nous faut donc reproduire le modèle de données sous la forme de la configuration Doctrine. Nous allons le faire au format XML. Voici les 3 objets métiers : user, tag et bookmark.

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/
doctrine-mapping"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://doctrine-project.org/schemas/orm/
doctrine-mapping https://www.doctrine-project.org/schemas/orm/
doctrine-mapping.xsd">
    <entity name="App\Domain\Bookmark\Model\Tag" table="tags">
        <unique-constraints>
            <unique-constraint columns="title" name="title" />
        </unique-constraints>
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>
        <field name="title" nullable="true" />
        <field name="created" type="datetime_immutable" nullable="true" />
        <field name="modified" type="datetime_immutable" nullable="true" />
        <many-to-many field="bookmarks" target-entity="App\Domain\Bookmark\
Model\Bookmark" mapped-by="tags">
            <cascade>
                <cascade-remove/>
            </cascade>
        </many-to-many>
    </entity>
</doctrine-mapping>
```

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/
doctrine-mapping"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://doctrine-project.org/schemas/orm/
doctrine-mapping https://www.doctrine-project.org/schemas/orm/
doctrine-mapping.xsd">
    <entity name="App\Domain\Bookmark\Model\User" table="users">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>
        <field name="email" />
        <field name="password" />
        <field name="created" type="datetime_immutable" nullable="true" />
        <field name="modified" type="datetime_immutable" nullable="true" />
        <one-to-many field="bookmarks" target-entity="App\Domain\Bookmark\
Model\Bookmark" mapped-by="user">
            <cascade>
                <cascade-remove/>
```

```
            </cascade>
        </one-to-many>
    </entity>
</doctrine-mapping>
```

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/
doctrine-mapping"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://doctrine-project.org/schemas/orm/
doctrine-mapping https://www.doctrine-project.org/schemas/orm/
doctrine-mapping.xsd">
    <entity name="App\Domain\Bookmark\Model\Bookmark" table="bookmarks">
        <indexes>
            <index columns="user_id" name="user_key" />
        </indexes>
        <id name="id" type="integer">
            <generator />
        </id>
        <field name="title" length="50" nullable="true" />
        <field name="description" nullable="true" length="2048" />
        <field name="url" length="2048" nullable="true" />
        <field name="created" type="datetime_immutable" nullable="true" />
        <field name="modified" type="datetime_immutable" nullable="true" />
        <many-to-one field="user" target-entity="User" inversed-by="bookmarks">
            <join-columns>
                <join-column name="user_id" nullable="false"/>
            </join-columns>
        </many-to-one>
        <many-to-many field="tags" target-entity="Tag" inversed-by="bookmarks">
            <join-table name="bookmarks_tags">
                <join-columns>
                    <join-column name="bookmark_id"/>
                </join-columns>
                <inverse-join-columns>
                    <join-column name="tag_id"/>
                </inverse-join-columns>
            </join-table>
        </many-to-many>
    </entity>
</doctrine-mapping>
```

Ces fichiers servent à indiquer à Doctrine comment les différentes propriétés des objets doivent être stockées en base de données, permettant de le transformer automatiquement vers la bdd, et quand on

récupère les données de la bdd. C'est d'ailleurs le sens de l'acronyme ORM : Object Relation Mapper.

Comme vous pouvez le voir, les objets que nous souhaitons faire connaître à Doctrine sont directement les objets métiers tel que nous les avons défini dans la couche modèle. Cela nous évitera de mettre en place un mécanisme de conversion entre entité Doctrine et objet métier, comme nous l'avons fait pour la partie Cakephp. Cependant, nous ne devrions pas mettre les définitions des champs de la table dans les fichiers domaine (annotation), car on va mélanger les couches domaine et infrastructure en faisant cela. Voilà pourquoi nous avons choisi le format XML pour ces définitions.

Pour être complet, il nous faut modifier le fichier SQL de base (côté Cakephp) pour correspondre au nommages de Doctrine et pour fixer des types mieux adaptés.

```
## configCakephp/schema/app.sql
…
CREATE TABLE bookmarks (
    …
    description TEXT,
    description VARCHAR(2048),
    url TEXT,
    url VARCHAR(2048),
    …
);

CREATE TABLE bookmarks_tags (
    …
    PRIMARY KEY (bookmark_id,tag_id),
    KEY IDX_CD7027B7BAD26311 (tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags (id),
    FOREIGN KEY bookmarks_tags_ibfk_1(tag_id) REFERENCES tags (id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks (id)
    FOREIGN KEY bookmarks_tags_ibfk_2(bookmark_id) REFERENCES bookmarks (id)
);
```

La difficulté à manipuler deux infrastructures comme nous le faisons là, c'est qu'elles doivent rester en cohérence dans leur connaissance du monde extérieur, et notamment de la base de données. Il faut décider laquelle des deux sera maître pour gérer le modèle en bdd, et laquelle suivra cette structure. Pour notre cas, nous garderons Cakephp comme responsable dans cette tâche.

Si vous avez bien suivi, nous avons mis en place une incompatibilité dans la configuration du mapping de l'objet bookmark. En effet, la propriété url de cet objet n'est pas une chaîne, mais un value-object. Il va nous falloir expliquer à Doctrine comment stocker et récupérer cette donnée. Pour cela, on va créer un nouveau type Doctrine.

```
namespace App\Symfony\Doctrine\Type;

use App\Domain\Bookmark\ValueObject\Url;
use Doctrine\DBAL\Platforms\AbstractPlatform;
use Doctrine\DBAL\Types\StringType;

class UrlType extends StringType
{
    public function convertToPHPValue($value, AbstractPlatform $platform): ?Url
    {
        if (!$value) {
            return null;
        }

        return Url::fromPersistedString($value);
    }

    public function convertToDatabaseValue($value, AbstractPlatform
$platform): ?string
    {
        if (!$value instanceof Url) {
            return null;
        }

        return $value->value;
    }

    public function getName(): string
    {
        return 'vo_url';
    }

    public function requiresSQLCommentHint(AbstractPlatform $platform): bool
    {
        return true;
    }
}
```

Cette classe est finalement assez claire. On définit un nouveau type qui

étend le type string, car on va bien le stocker sous la forme d'une chaîne de caractères. On a là deux méthodes essentielles : comment on la convertit pour php, et comment on la convertit pour la bdd. Simple. Le reste est détail technique. Précisons à Doctrine que ce nouveau type existe.

```
## config/packages/doctrine.yaml
doctrine:
    dbal:
        url: '%env(resolve:DATABASE_URL)%'
        types:
            vo_url: App\Symfony\Doctrine\Type\UrlType
    …
```

Puis utilisons le type dans notre mapping `config/entity/bookmark/Bookmark.orm.xml`

```
        <field name="url" length="2048" nullable="true" />
        <field type="vo_url" name="url" length="2048" nullable="true" />
```

Pour aider Doctrine à s'y retrouver, il faut aussi ajouter un commentaire à la colonne dans la bdd.

```
#configCakephp/schema/app.sql
CREATE TABLE bookmarks (
    …
    url VARCHAR(2048),
    url VARCHAR(2048) COMMENT '(DC2Type:vo_url)',
    …
);
```

On peut ajouter autant de types que l'on souhaite, ce qui nous permet de passer vraiment toutes les propriétés sous la forme de value-object. Et pour les value-object qui détiennent plusieurs données, comme une adresse postale, Doctrine propose un concept d'(embeddable)[https://www.doctrine-project.org/projects/doctrine-orm/en/2.13/tutorials/embeddables.html] qui permet de le répartir dans plusieurs champs de bdd.

Nous voilà avec une configuration doctrine complète. Commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/472a2ea7dcb8525c4ab946b19b54c8fc84d6b33c

On peut ajouter la commande suivante à notre série de commande de validation de la qualité `docker-compose exec web bin/console do:sc:va`

7.3. Because I'm API

Il est maintenant possible d'exposer nos objets sur une Api contre un peu de configuration.

```yaml
## config/packages/api_platform.yaml
api_platform:
  defaults:
    pagination_items_per_page: 20
    pagination_maximum_items_per_page: 100
  collection:
    pagination:
      page_parameter_name: page
      items_per_page_parameter_name: itemsPerPage
  mapping:
    paths:
      - '%kernel.project_dir%/config/api_platform/bookmark'
  formats:
    jsonld: [ 'application/json-ld' ]
    html: [ 'text/html' ]
    json: [ 'application/json' ]
    form: [ 'multipart/form-data' ]
  patch_formats:
    json: [ 'application/merge-patch+json' ]
  swagger:
    versions: [ 3 ]
    api_keys:
      apiKey:
        name: Authorization
        type: header
```

On reste là très proche de la configuration par défaut d'Api-Platform. Le point notable est uniquement le chemin pour trouver les différents mapping des objets exposés par l'Api. Voici. celui d'un bookmark.

```
## config/api_platform/bookmark/bookmark.yaml
App\Domain\Bookmark\Model\Bookmark:
    operations:
        ApiPlatform\Metadata\GetCollection: ~
        ApiPlatform\Metadata\Get: ~
```

On définit ici les deux opérations possibles : obtenir la liste des bookmarks, et obtenir un bookmark sur la base de son id.

On peut lancer la requête immédiatement, l'interface de swagger n'est peut-être pas la plus confortable mais elle est d'ores et déjà disponible sur http://0.0.0.0:9050/api/docs. On se heurte alors à une erreur : `Cannot assign Doctrine\\ORM\\PersistentCollection to property App\\Domain\\Bookmark\\Model\\User::$bookmarks of type array`. La raison en est simple, Doctrine ne travaille pas avec des tableaux, mais avec des objets qu'il nomme collections, pour les listes d'entités. C'est très pratique pour gérer les paginations. Imaginez que 100.000 tags soient associés à notre bookmark : la simple requête serait insupportable au serveur.

Bref, il n'y a pas vraiment de solutions de contournement, et nous allons faire une entorse à notre DDD en définissant cette collection comme valide pour les propriétés des objets métier.

```
namespace App\Domain\Bookmark\Model;

use App\Domain\Bookmark\ValueObject\Url;
use DateTimeImmutable;
use Doctrine\Common\Collections\Collection;

class Bookmark
{
    …
    /** @var array<Tag> */
    public array $tags;
    /** @var array<Tag>|Collection<int, Tag> */
    public array|Collection $tags;
    …
}
```

On fait de même avec les propriétés `App\Domain\Bookmark\Model\Tag::bookmarks` puis `App\Domain\Bookmark\Model\User::bookmarks`. Puis on relance. Nouvelle erreur : `A circular reference has been`

`detected when serializing the object of class \"Proxies\\__CG__\\App\\Domain\\Bookmark\\Model\\User\" (configured limit: 1)`.. La raison en est tout simple : il recherche un bookmark, qui appartient à un utilisateur, qui a plusieurs bookmark, dont chacun appartient à un utilisateur, qui a plusieurs ... etc. etc. Vous voyez la boucle. Pour y remédier, nous allons en même temps améliorer l'api. En fait on aurait envie de pouvoir lui dire que champ doit être affiché en fonction du contexte. On va va définir cela au niveau de la sérialisation dont le boulot est de transformer un objet métier dans sa représentation en une chaîne de caractères. Dans notre cas, il s'agit de JSON. Définissons les propriétés accessibles depuis un « groupe » de sérialisation.

```
## config/serializer/bookmark.yaml
App\Domain\Bookmark\Model\Bookmark:
    attributes:
        title:
            groups: ['bookmark:read']
        url:
            groups: ['bookmark:read']
        description:
            groups: ['bookmark:read']
        user:
            groups: ['bookmark:read']
        tags:
            groups: ['bookmark:read']
```

```
## config/serializer/user.yaml
App\Domain\Bookmark\Model\User:
    attributes:
        email:
            groups: ['bookmark:read']
```

```
App\Domain\Bookmark\Model\Tag:
    attributes:
        title:
            groups: ['bookmark:read']
```

Puis on utilise ce groupe.

```
App\Domain\Bookmark\Model\Bookmark:
    normalizationContext:
        groups: ['bookmark:read']
    operations:
```

```
        ApiPlatform\Metadata\GetCollection: ~
        ApiPlatform\Metadata\Get: ~
```

Nous voilà équipés d'un outillage pour personnaliser très finement ce que nous voudrons faire sortir via l'Api.

Voyons le résultat lorsque nous faisons la requête sur notre point d'Api : http://0.0.0.0:9050/api/bookmarks?page=1


```
{
  "@context": "/api/contexts/Bookmark",
  "@id": "/api/bookmarks",
  "@type": "hydra:Collection",
  "hydra:member": [
    {
      "@id": "/api/bookmarks/1",
      "@type": "Bookmark",
      "title": "CakePHP",
      "url": [],
      "description": "Build Fast, Grow Solid",
      "user": {
        "@type": "User",
        "@id": "/api/.well-known/genid/e8eb4f8fc151ce96ea2e",
        "email": "user@example.com"
      },
      "tags": [
        {
          "@type": "Tag",
          "@id": "/api/.well-known/genid/adb9b627219a0b6f37c6",
          "title": "development"
        },
        …
     ],
    },
    …
  ],
  "hydra:totalItems": 5
}
```


Si vous regardez bien, il y a un problème ici : l'url est donnée sous la forme d'un tableau vide. C'est dû au serializer qui n'a pas su comment transformer la value-object Url en représentation valide pour la transmettre. Nous allons donc lui expliquer comment la sérialiser.

```php
namespace App\Symfony\Serializer;

use App\Domain\Bookmark\ValueObject\Url;
use Exception;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

final class UrlSerializer implements NormalizerInterface
{
    /**
     * {@inheritDoc}
     */
    public function normalize($object, $format = null, array $context = [])
    {
        if (!$object instanceof Url) {
            throw new Exception('Invalid Type. Url Required.');
        }

        return $object->value;
    }

    /**
     * {@inheritDoc}
     */
    public function supportsNormalization($data, $format = null): bool
    {
        return $data instanceof Url;
    }
}
```

Puis il nous faut déclarer cette classe au composant de serialisation de Symfony. Nous allons le faire de manière générique en lui demandant d'enregistrer toute classe qui implémente `NormalizerInterface`.

```yaml
# config/services.yaml
services:

    _instanceof:
        Symfony\Component\Serializer\Normalizer\NormalizerInterface:
            tags: [ 'serializer.normalizer' ]
```

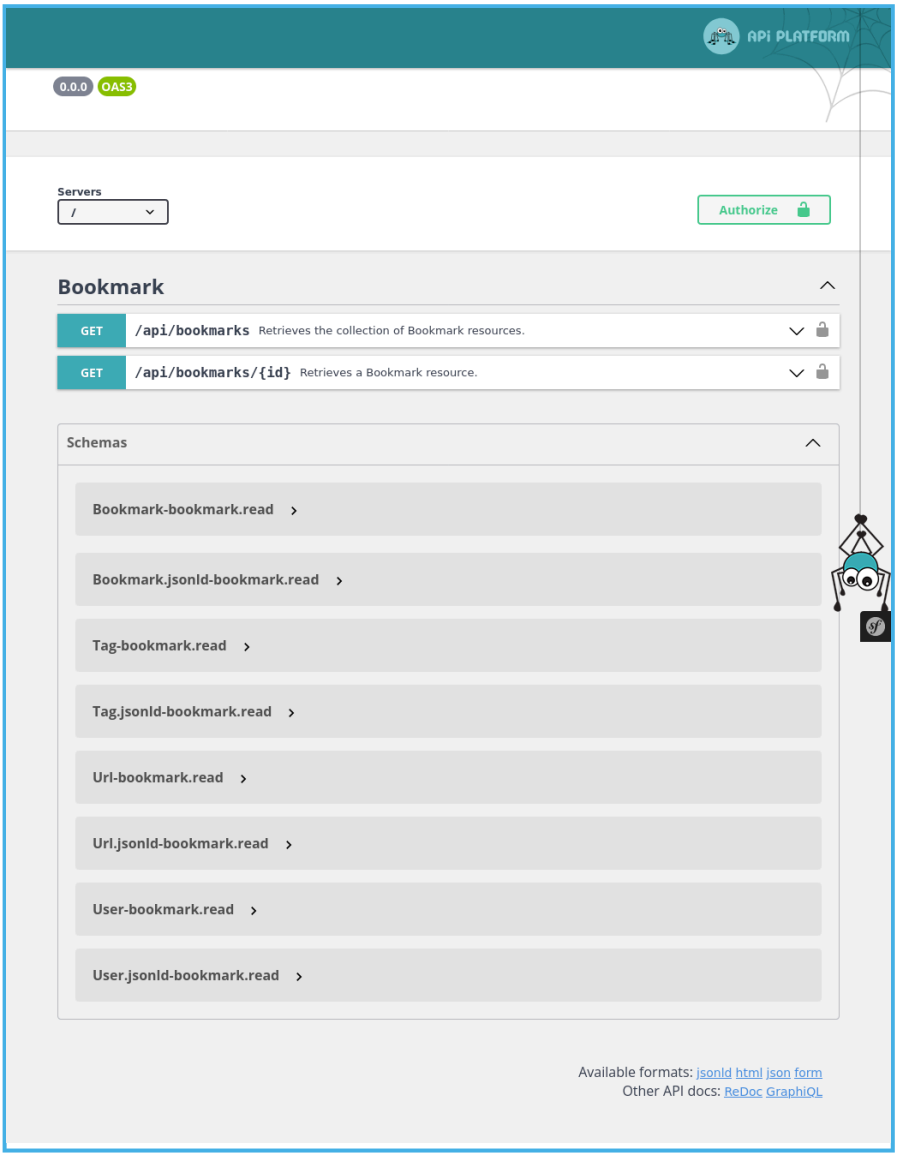
Il faudra en faire de même pour tous les value-object, mais on pourra avantagement créer une interface pour les value object qui sont basée sur une seule valeur, et utiliser le même normaliseur.

```
{
  "@context": "/api/contexts/Bookmark",
  "@id": "/api/bookmarks",
  "@type": "hydra:Collection",
  "hydra:member": [
    {
      "@id": "/api/bookmarks/2",
      "@type": "Bookmark",
      "title": "Mozilla",
      "url": "http://mozilla.org",
      "description": "Internet for People, Not Profit",
      "user": {
        "@type": "User",
        "@id": "/api/.well-known/genid/66f826e80d3bc5f817ed",
        "email": "user@example.com"
      },
      "tags": [
        {
          "@type": "Tag",
          "@id": "/api/.well-known/genid/3bd8be0d138e12f11261",
          "title": "servo"
        },
        …
      ]
    },
    …
  ],
  "hydra:totalItems": 5
}
```


D'un point de vue DDD, nous avons réalisé cette action sans jamais descendre explicitement dans la couche Domaine, nous n'avons pas utiliser le point d'entrée GetBookmark, mais nous allons laisser Doctrine faire ce travail pour nous. On peut tout à fait voir cette démarche comme contraire au DDD, mais c'est un compromis que nous assumons.

DDD La couche domaine étant indépendante, nous sommes en mesure de la faire utiliser par plusieurs application de nature très différentes comme cette API ou le fullstack de Cakephp.

Finalement, nous avons mis en place cette API sans grand effort ! Presque uniquement par de la configuration. Voilà un résultat tout à fait encourageant !



Le commit de ces changements : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/61d1063526dd389f2d9cb8c92780468dab1d821c

7.4. Un point d'API en écriture

Ajoutons mettons la possibilité de modifier un bookmark depuis l'API.

```
# config/api_platform/bookmark/bookmark.yaml

App\Domain\Bookmark\Model\Bookmark:
    …
    operations:
        …
        ApiPlatform\Metadata\Patch:
            input: App\Application\UpdateBookmark\UpdateBookmarkInput
            processor: ApiPlatform\Symfony\Messenger\Processor
```

Nous lui disons ici que nous ajoutons une action de PATCH, c'est-à-dire de mettre à jour certaines données du bookmark. Pour cela nous lui donnons un input, qu'Api-Platform va tâcher de construire sur la base des paramètres passées par la requête HTTP. Puis nous lui précisons le processeur, la classe qui sera en mesure traiter la demande.

Nous faisons le choix de passer par le système de message de Symfony. Ce système permet d'émettre des messages, sous la forme d'une classe php (ce sera notre input), qui est ensuite intercepté par une autre classe capable de les traiter (notre handler).

```
docker-compose exec web composer require symfony/messenger
```

Il nous faut maintenant définir que nos handler sont en mesures de traiter des messages envoyés via le messenger. Pour le faire de manière gloable, nous allons passer par une interface.

```
namespace App\Application\UpdateBookmark;

use App\Application\Handler;
…

~~class UpdateBookmarkHandler~~
class UpdateBookmarkHandler implements Handler
```

```
{
    …
}
```

L'interface est vide, elle nous sert juste a bien identifier nos affaires.

```
namespace App\Application;

interface Handler
{
}
```

Et maintenant, prévenons Symfony qu'il peut utiliser nos handler pour son messenger.

```
services:
    …
    _instanceof:
        …
        App\Application\Handler:
            tags: [ 'messenger.message_handler' ]
```

À présent, il nous faut implémenter côté infrastructure symfony l'outillage demandé par le domaine. Commençons par les repository.

```
namespace App\Symfony\Repository;

use App\Domain\Bookmark\Model\Bookmark;
use App\Domain\Bookmark\Repository\BookmarkRepository as
DomainBookmarkRepository;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class BookmarkRepostiory extends ServiceEntityRepository implements
DomainBookmarkRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Bookmark::class);
    }
```

```
    public function findById(int $id): ?Bookmark
    {
        return $this->find($id);
    }

    public function persist(Bookmark $bookmark): void
    {
        $this->getEntityManager()->persist($bookmark);
        $this->getEntityManager()->flush();
    }
}
```

```
namespace App\Symfony\Repository;

use App\Domain\Bookmark\Model\Tag;
use App\Domain\Bookmark\Repository\TagRepository as DomainTagRepository;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class TagRepostiory extends ServiceEntityRepository implements
DomainTagRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Tag::class);
    }

    public function findByTitle(string $title): ?Tag
    {
        return $this->findOneBy(['title' => $title]);
    }
}
```

Il va nous falloir implémenter l'interface pour retrouver l'utilisateur courant, mais puisque nous n'avons pas de système de sécurité côté symfony pour le moment, on va juste retourner null.

```
namespace App\Symfony\Security;

use App\Domain\Bookmark\Context\CurrentUserProvider as
DomainCurrentUserProvider;
use App\Domain\Bookmark\Model\User;

class CurrentUserProvider implements DomainCurrentUserProvider
```

```
{
    public function getCurrentUser(): ?User
    {
        return null;
    }
}
```

Pour faire fonctionner notre modification, il va falloir autoriser que la modification soit faite par un anonyme.

```
namespace App\Domain\Bookmark\Validator;

class BookmarkUpdaterValidator
{
    public function validate(Bookmark $bookmark): array
    {
        …
        if (null === $currentUser || $currentUser->id !== $bookmark->user->id)
{
        if (null !== $currentUser && $currentUser->id !== $bookmark->user->id)
{
        …
    }
}
```

À présent, précisons à Doctrine qu'un tage peut être créé si un nouveau tag associé à un bookmark est identifié.

```
        <many-to-many field="tags" target-entity="Tag" inversed-by="bookmarks">
            <cascade>
                <cascade-persist />
            </cascade>
            …
        </many-to-one>
```

C'est parti pour faire une nouvelle requête.

```
curl -X 'PATCH' \
  'http://0.0.0.0:9050/api/bookmarks/1' \
  -H 'accept: application/json-ld' \
  -H 'Content-Type: application/merge-patch+json' \
```

```
  -d '{
  "id": 1,
  "title": "Nouveau nom",
  "url": "http://example.com",
  "description": "Juste du blabla",
  "tagsTitle": [
    "Nouveau tag"
  ]
}'
```

Et voici le contenu de la réponse en 200.

```
{
  "@context": "/api/contexts/Bookmark",
  "@id": "/api/bookmarks/1",
  "@type": "Bookmark",
  "title": "Nouveau nom",
  "url": "http://example.com",
  "description": "Juste du blabla",
  "user": {
    "@type": "User",
    "@id": "/api/.well-known/genid/a14f9cfdaf0001fcfb0a",
    "email": "user@example.com"
  },
  "tags": [
    {
      "@type": "Tag",
      "@id": "/api/.well-known/genid/8274830f19af4a9f57c6",
      "title": "Nouveau tag"
    }
  ]
}
```

Le patch fonctionne parfaitement :) Voici le commit : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/04dfcff3d279619c4cb216f59975b3cc826b0eb0

Renvoyer l'élément modifié ou juste une réponse 201 est un choix technique à faire. Les deux présentent leurs avantages et leurs inconvénients.

7.5. Donner de bons retours d'Api

Testons maintenant de transmettre cette même requête, mais avec une erreur.

```
curl -X 'PATCH' \
  'http://0.0.0.0:9050/api/bookmarks/1' \
  -H 'accept: application/json-ld' \
  -H 'Content-Type: application/merge-patch+json' \
  -d '{
  "id": 1,
  "title": "Nouveau nom",
  "url": "pas d'url ici",
  "description": "Juste du blabla",
  "tagsTitle": [
    "Nouveau tag"
  ]
}'
```

Et voici le contenu de la réponse en 500.

```
{
    "@context":"\/api\/contexts\/Error",
    "@type":"hydra:Error",
    "hydra:title":"An error occurred",
    "hydra:description":"Errors occured with your request",
    …
}
```

Voilà une erreur qui n'est pas très explicite. L'exécution a été stoppée par la levée de l'exception. Pour pouvoir rendre de meilleures erreurs, nous allons nous interdire de lever des exceptions depuis les handler, mais renvoyer un objet qui a collecté ces erreurs à la place.

```
namespace App\Application\UpdateBookmark;

use App\Domain\Bookmark\Exception\ViolationCollectionException;
use App\Domain\Bookmark\Violation\ViolationCollector;

class UpdateBookmarkHandler implements Handler
{
```

```
    …
    public function __invoke(
        UpdateBookmarkInput $input
    ): Bookmark {
    ): Bookmark|ViolationCollector {
        …
        if (count($errors)) {
            throw new ViolationCollectionException('Errors occured with your
request', $errors);
            return new ViolationCollector($errors);
        }
        …
    }
}
```

Avec un peu de recul, nous sommes en train de passer d'un tableau de string que nous nommons «erreur», à un objet collection qui va détenir une liste de violations, qui pourra détenir plus de valeur métier. Créons cette violation d'abord.

```
namespace App\Domain\Bookmark\Violation;

class Violation
{
    public function __construct(
        public readonly string  $message,
        public readonly ?string $propertyPath = null,
    ) {
    }
}
```

Puis le collecteur qui portera ces violations.

```
namespace App\Domain\Bookmark\Violation;

class ViolationCollector
{
    /**
     * @var array
     */
    private array $violations = [];

    public function collect(Violation $violation): void
```

```
    {
        $this->violations[] = $violation;
    }

    public function hasViolations(): bool
    {
        return count($this->violations) > 0;
    }

    /**
     * @return Violation[]
     */
    public function getViolations(): array
    {
        return $this->violations;
    }
}
```

Nous pouvons maintenant modifier toutes nos création d'erreurs pour leur apporter bien plus de sens.

```
namespace App\Application\UpdateBookmark;

use App\Domain\Bookmark\Violation\Violation;
use App\Domain\Bookmark\Violation\ViolationCollector;

class UpdateBookmarkValidator
{
    public function __construct (
        private readonly ViolationCollector $violationCollector,
    ) {
    }

    public function validate(
        UpdateBookmarkInput $input
    ~~): array {~~
    ): void {
        ~~$violations = [];~~
        if (mb_strlen($input->title) < 3) {
            ~~$violations[] = 'Title must be at last 3 char long.';~~
            $this->violationCollector->collect(new Violation('Title must be at
last 3 char long.', 'title'));
        }
        if (mb_strlen($input->title) > 1024) {
```

```
            $violations[] = 'Title cannot be more than 1024 char long.';
            $this->violationCollector->collect(new Violation('Title cannot be
more than 1024 char long.', 'title'));
        }

        return $violations;
    }
}
```

Puis on fait de même pour les autres validateurs. Toutes les violations se trouvent ainsi rassemblées en un seul endroit.

On peut mettre à jour notre handler.

```
namespace App\Application\UpdateBookmark;

…
use App\Domain\Bookmark\Violation\Violation;
use App\Domain\Bookmark\Violation\ViolationCollector;

class UpdateBookmarkHandler implements Handler
{
    public function __construct(
        …
        private readonly ViolationCollector $violationCollector,
    ) {
    }

    public function __invoke(
        UpdateBookmarkInput $input,
    ): Bookmark|ViolationCollector {
        $errors = $this->inputValidator->validate($input);
        $this->inputValidator->validate($input);
        $bookmark = $this->bookmarkRepository->findById($input->id);
        if (!$bookmark) {
            $errors[] = 'Bookmark does not exists.';
            $this->violationCollector->collect(new Violation('Bookmark does
not exists.'));
        } else {
            $errors = array_merge($errors,
$this->updateValidator->validate($bookmark));
            $this->updateValidator->validate($bookmark);
        }
        …
```

```
        if (count($errors)) {
        if ($this->violationCollector->hasViolations()) {
            throw new ViolationCollectionException('Errors occured with your
request', $errors);
            return $this->violationCollector;
        }
        …
    }
}
```

Quand tout est ok, on relance un test de modification en curl. Et on obtient un tableau vide ! Cela vient du fait que le serializer n'a pas su comment donner une version de sortie de la classe ViolationCollector. Nous allons donc lui expliquer.

```
namespace App\Symfony\Serializer;

use ApiPlatform\Symfony\Validator\Exception\ValidationException;
use App\Domain\Bookmark\Violation\Violation;
use App\Domain\Bookmark\Violation\ViolationCollector;
use LogicException;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;
use Symfony\Component\Validator\ConstraintViolation;
use Symfony\Component\Validator\ConstraintViolationInterface;
use Symfony\Component\Validator\ConstraintViolationList;

final class ViolationCollectorSerializer implements NormalizerInterface
{
    /**
     * {@inheritDoc}
     */
    public function normalize($object, $format = null, array $context = [])
    {
        if (!$object instanceof ViolationCollector) {
            throw new LogicException();
        }

        if (!$object->hasViolations()) {
            throw new \LogicException('ViolationCollection cannot be empty');
        }

        $constraintViolations = array_map(static function (Violation
$violation): ConstraintViolationInterface {
            return new ConstraintViolation(
```

```
                message: $violation->message,
                messageTemplate: $violation->message,
                parameters: [],
                root: null,
                propertyPath: $violation->propertyPath,
                invalidValue: null,
                code: md5($violation->propertyPath.$violation->message),
            );
        }, $object->getViolations());

        throw new ValidationException(new
ConstraintViolationList($constraintViolations));
    }

    /**
     * {@inheritDoc}
     */
    public function supportsNormalization($data, $format = null): bool
    {
        return $data instanceof ViolationCollector;
    }
}
```

Ici on utilise la capacité d'Api-Platform a récupérer les erreurs des contraintes de Symfony, pour en faire un affichage correct en sortie. Nous transformons donc nos violations en contraintes Symfony. Petit test.

```
curl -X 'PATCH' \
  'http://0.0.0.0:9050/api/bookmarks/1' \
  -H 'accept: application/json-ld' \
  -H 'Content-Type: application/merge-patch+json' \
  -d '{
  "id": 1,
  "title": "Nouveau nom",
  "url": "http://example.com",
  "description": "Juste du blabla",
  "tagsTitle": [
    "Nouveau tag"
  ]
}'
```

Et voici le contenu de la réponse en 400.

```
{
  "@context": "/api/contexts/ConstraintViolationList",
  "@type": "ConstraintViolationList",
  "hydra:title": "An error occurred",
  "hydra:description": "url: Invalid URL",
  "violations": [
    {
      "propertyPath": "url",
      "message": "Invalid URL",
      "code": "e6bd3d3d1bd3b70daf08e5bed1b6bad5"
    }
  ]
}
```

Voilà une réponse tout à fait exploitable, même pour une application frontend !

Malgré sa cure de rafraîchissement, en regardant à nouveau le Handler, je dois avouer qu'il fait trop de tâches. Je ne peux résister de mettre en place une factory pour créer les value-object.

```
namespace App\Domain\Bookmark\ValueObject;

use App\Domain\Bookmark\Violation\Violation;
use App\Domain\Bookmark\Violation\ViolationCollector;
use Exception;

class ValueObjectFactory
{
    public function __construct(
        private readonly ViolationCollector $violationCollector,
    ) {
    }

    public function makeFromString(string $class, string $value, string
$propertyPath): ?ValueObjectBasedOnString
    {
        if (!class_exists($class)) {
            throw new Exception(sprintf('Cannot create value object %s',
$class));
        }
        $interfaces = class_implements($class) ?: [];
        if (!in_array(ValueObjectBasedOnString::class, $interfaces)) {
            throw new Exception(sprintf('Cannot create value object %s',
```

```
$class));
        }

        try {
            return $class::fromString($value);
        } catch (InvalidValueException $exception) {
            $this->violationCollector->collect(new
Violation($exception->getMessage(), $propertyPath));
        }

        return null;
    }
}
```

Et on modifie le handler pour y faire appel.

```
namespace App\Application\UpdateBookmark;

use App\Domain\Bookmark\ValueObject\ValueObjectFactory;

class UpdateBookmarkHandler implements Handler
{
    public function __construct(
        …
        private readonly ValueObjectFactory $valueObjectFactory,
    ) {
    }

    public function __invoke(
        UpdateBookmarkInput $input,
    ): Bookmark|ViolationCollector {
        …
        try {
            $url = Url::fromString($input->url);
        } catch (InvalidValueException $exception) {
            $url = null;
            $this->violationCollector->collect(new
Violation($exception->getMessage(), 'url'));
        }
        $url = $this->valueObjectFactory->makeFromString(Url::class,
$input->url, 'url');
        …
    }
}
```

Notre value object implémente maintenant cette simple interface.

```
namespace App\Domain\Bookmark\ValueObject;

interface ValueObjectBasedOnString
{
    public static function fromString(string $value): self;
}
```

Il nous faut aussi mettre à jour le container de services côté Cakephp pour injecter ces nouvelles dépendances.

```
namespace App\Controller\Component;

class ContainerComponent extends Component
{
    public function __construct(ComponentRegistry $registry, array $config =
[])
    {
        …
        $this->container[ViolationCollector::class] = new ViolationCollector();
        $this->container[ValueObjectFactory::class] = new
ValueObjectFactory($this->container[ViolationCollector::class]);
        $this->container[UpdateBookmarkValidator::class] = new
UpdateBookmarkValidator($this->container[ViolationCollector::class]);
        $this->container[UpdateBookmarkValidator::class] = new
UpdateBookmarkValidator();
        $this->container[BookmarkUpdaterValidator::class] = new
BookmarkUpdaterValidator($this->CurrentUserProvider);
        $this->container[BookmarkUpdaterValidator::class] = new
BookmarkUpdaterValidator($this->CurrentUserProvider,
$this->container[ViolationCollector::class]);
        $this->container[UpdateBookmarkHandler::class] = new
UpdateBookmarkHandler(
            …
            $this->container[ViolationCollector::class],
            $this->container[ValueObjectFactory::class],
        );
    }
    …
}
```

Au final, nous voici un handler beaucoup plus propre et compréhensible, avec une première partie sur la validation comprenant la création des

value objects et une deuxième sur la mise à jour à proprement parler.

En voici le commit avec des modifications précises : https://github.com/vibby/cakephp-bookmarker-tutorial/commit/11ba291131b4dbc1c23c9e526ce5ed83011805bc

Je pourrais encore trouver de nombreux axes d'améliorations, mais je crois que nous allons nous arrêter ici :) J'espère que vous avez apprécié le voyage !

LE MOT DE LA FIN

Que de chemin parcouru ! Notre application est maintenant opérationnelle et bien découpée en terme de responsabilités. Clairement, cette approche est un compromis entre le respect des principes du DDD et le pargmatisme d'un projet concret préexistant. Elle permet de limiter au maximum les dépendances entre les briqus logiciels, qui est un gage de meilleure tenue dans le temps par une maintenabilité grandement améliorée.

Cependant, un découpage plus important apporte également nécessairement plus de code et certains pourait se sentir perdus dans cette organisation. Autre point négatif : la mise en place d'un tel paradigme nécessite l'implication de tous les acteurs du projet, y compris la partie « métier ». Mais le gain est énorme pour une application qui a pour objectif de vivre longtemps. D'ailleurs, depuis mon experience, toute l'équipe a une meilleure confiance dans le projet dans ce type d'architecture. Chez Troopers, nous avons travaillés sur 2 projets importants avec ce type de conversion.

Si vous découvrez le DDD, je vous propose de commencer par le bouquin d'Eric Evans à l'origine de toute l'aventure : **Domain-Driven Design: Tackling Complexity in the Heart of Software**. https://www.chasse-aux-livres.fr/prix/0321125215/domain-driven-design-eric-evans

Pour aller plus loin sur le sujet, je vous conseille l'excellent article d'Alex So Yes : https://alexsoyes.com/ddd-domain-driven-design/#utiliser-ddd-dans-son-projet-tout-de-suite

Vous avez un héritage logiciel ? Vous aimeriez en valoriser la partie métier, et moderniser son infrastructure ? Assurément cette méthode est faite pour vous, en séparant l'un de l'autre !