21. Model and Repositories aiming to replace the usages of legacy ObjectModel

Date: 2023-10-27

Status

Rejected

Decision

This proposal will not be applied, the main arguments against it are:

- the fear of adding a new extra layer that would confuse the community more than it helps it
- the cost/time it would take to apply this on all the entities in place of other more valuable topics
- adding this extra layer would indeed hide the legacy core classes, but it doesn't remove their usage, it mostly hides it

Instead of going into that direction it was decided that it would be more valuable for the project to start migrating the core classes to Doctrine entities, an ADR has been opened to define the strategy to do this https://github.com/PrestaShop/ADR/pull/37

Context

Today, in most cases, the way to get a data from our database is to rely on legacy ObjectModel classes. This model has some advantages and some drawbacks but already knwo we don't plan on keeping it in the future (at least not in its exact current form). Yet we keep working on features, services, CQRS commands or miscellaneous data providers that need to access to this data. We also have other alternatives of course (Doctrine entities, legacy DB queries, DBAL queries) but the problem stays the same all the solutions we have now rely on a code that is bound to disappear or be modificed, thus everytime we create a code relying on these ObjectModels we create a new technical debt. It may not be a big one, but a technical debt nonetheless since we'll have to remove/replace this code at some point.

The purpose of this ADR is to introduce a new access to our data, one that is independent from its actual implementation (ObjectModel, Doctrine, DBAL, ...), we propose to add some new interfaces that represent our Model data, it's DATA only these interfaces don't have domain or business logic, they are simply a mean to acces our data. We chose interfaces because they have the advantage of being independent of the implementation, it includes the different ways to access the DB, but it goes further we could even imagine data accessed on a remote API. With an interface all this internal code doesn't matter any more as long as the interface is respected.

Summary this ADR proposes to add new **Model interfaces** represent our data model, it also includes some **Repository** interfaces each one being associated to its Model interface. We will define and structure with as much details as possible the structure, architecture and naming convention that will be imposed on these interfaces (and part of their implementation) so that these questions don't need to be thought about during implementation and/or review.

Limitations of this ADR

- we didin't take into account the possible relationships between models (except for multi shop which is a special case), this could be immproved in a second ADR, but for starters we will focus only the model "basic fields" (aka their columns in the DB).
- we focus on read only for now, we plan on extending this principle to write/update/delete operations but it's more complex so it will also be dealt in another ADR.

1. Model Interfaces

Take all classes based on ObjectModel and turn them into a **Model interface** representing each properties and database columns

Generic interface (always translated)

Object with multilang properties have getters for those field which return only one string already localized (the language id is specified when fetching the object).

```
interface ContactInterface {
  public function getContactId(): int;
  public function getEmail(): int;

  // Localized getters
  public function getName(): string;
  public function getDescription(): string;
}
```

Localized interface (mostly for CRUD)

Some use cases require having all the data for all languages (for edition, or multi lang components), then the multilang fields are returned as an associative array (language ID used as the key). They don't include the mono-lang getter.

```
interface LocalizedContactInterface {
    public function getContactId(): int;
    public function getEmail(): int;

    /**
     * @return array<int, string> Associative array, the key is languageId and the value is the localized value.
     */
    public function getLocalizedNames(): array;
```

```
    public function getLocalizedDescriptions(): array;
}
```

Scalar values as much as possible

The fields/getters should return scalr values as much as possible, we don't use ValueObject so that these interfaces are by default compatible with as many use cases as possible by remaining simple (easier for serialization). `DecimalNumber` is considered as a scalar value since it's better that a native float, actually it should even be used as mandatory type for all decimal values (integers shoudl be int though).

For similar reasons even if it's not possible to strictly type arrays we will rely on arrays instead of collection to simplify the implementation (maybe one day PHP language will evolve and handle typed array, then we may be able to improve this).

2. Repositories

Get entity by ID:

- multi lang entities MUST specify a languageId
- multi shop entities which content can vary depending on shop MUST specify a shopId (it it's only a basic association without content modification a simple `getAssociatedShopIds()` is enough and the `shopId` parameter is not necessary)
- multi shop entities which have more complex multi shop behaviour MAY have a getter based on ShopConstraint
- entities which are neither multi shop nor multi lang don't need these parameters of course

Two repositories are needed for multi lang interfaces since they don't return the same interface. However for multi shop entities the shopId is always mandatory.

```
interface ContactRepositoryInterface {
    public function getContact(int contactId, int languageId, int shopId): ContactInterface;
}

interface LocalizedContactRepositoryInterface {
    /**
     * Returns ALL languages present in the database.
     *
     * @param contactId
     * @param shopId
     *
     * @return LocalizedContactInterface
     */
    public function getLocalizedContact(int contactId, int shopId): LocalizedContactInterface;

    /**
     * @param contactId $
     * @param shopId $
     * @param languages List of filtered languages
     *
     * @return LocalizedContactInterface
     */
    public function getLocalizedContactWithLanguages(int contactId, int shopId, int languages): LocalizedContactInterface;
}
```

Exceptions

Not found Exception When the entity is not found an exception should be thrown `CurrencyNotFoundException ContactNotFoundException`

Not found exception MUST extend the associated Model Exception (`CurrencyException`), but each type of exception MAY implement a generic Exception interface.

(Note we already implemented many DomainException for CQRS, the question here is should we capitalize on them, refactor them? If not a mapping Model Exception -> Domain Exception may have to be implemented for existing commands which may be troublesome)

```
class CurrencyNotFoundException extends CurrencyException implements EntityNotFoundExceptionInterface {
}
```

Invalid argument Exception The provided argument should follow some valid values (like being positive ID), when an invalid argument is provided we should throw an `InvalidArgumentException` that extends the base `ModelException`.

Get byXYZ

```
interface CurrencyRepositoryInterface {
    public function getCurrency(int currencyId, int languageId, int shopId): CurrencyInterface;
    public function getCurrencyByIsoCode(int isoCode, int languageId, int shopId): CurrencyInterface;

    public function getCurrencyBy(array $criteria, int languageId, int shopId): CurrencyInterface;
}
```

When more than one entity is found (can happen with getByCriteria) an exception should be thrown `NonUniqueCurrencyException` that extends `CurrencyException` and implements `NonUniqueEntityExceptionInterface`

List / search / filter (return IDs or full objects)

Decision must be taken between

- return an array of interfaces (we lose the strong type, but no need to implement a collection for each entity, MAYBE more performant?)
- return a collection of interfaces

Decision must be taken between

- findAll
- getAll

```
interface CurrencyRepositoryInterface {
    /**
     * @return CurrencyInterface[]
     */
    public function findAll(): array;
    public function getAll(): CurrencyCollectionInterface;

    public function findBy(array $filters = [], array $sortBy = [], ?int $limit = null, ?int $offset = null): array;
}
```

3. DBAL queries VS Object models VS Doctrine entities

- In all cases we will have a base implementation in `PrestaShopBundle\Model\Repository` that is preferably based on DBAL
  - the DBAL queries are implemented inside this repository and passed into DTOs
  - you will need to create a DTO in `PrestaShopBundle\Model` that implement the interface

Specific cases based on legacy:

- If the implementation is based on `ObjectModel` you create an `ObjectModelRepository` in `Adapter/MyEntityDomain/Repository/MyEntityRepository`
  - the `ObjectModel` must implement the Model interface
  - the `ObjectModelRepository` is passed as a dependency into the `PrestaShopBundle\Repository` and is used a proxy so it doesn't need to implement the interface itself
- If the implementation is based on a Doctrine Entity you create an `EntityRepository` in `PrestashopBundle/Entity/Repository/MyEntityRepository`
  - the Doctrine entity must implement the interface
  - the Doctrine repository is passed as a dependency into the PrestaShopBundle\Repository and is used a proxy so it doesn't need to implement the interface itself

DBAL method example

```
namespace PrestaShopBundle\Model\Repository;

class AccessRepository implements AccessRepositoryInterface
{
    public __contruct(Connection $connection) {
        $this->connection = $connection;
    }

    /**
     * @return RoleInterface[]
     */
    public function getRoles(): array
    {
        // Or implement it via DBAL query
        $queryData = $this->connection->createQueryBuilder()
            ...
        ;

        // Parse data and create DTO objects that implement the interface

        return $dtos;
    }
}
```

ObjectModel example

```
namespace PrestaShopBundle\Model\Repository;

use PrestaShop\PrestaShop\Adapter\Product\Repository\ProductRepository as AdpaterProductRepository;
```

```
class ProductRepository implements ProductRepositoryInterface
{
    private $productRepository;

    public __contruct(AdapterProductRepository $productRepository) {
        $this->productRepository = $productRepository;
    }

    public function getProduct(int $productId, int $languageId, int $shopId): LocalizedProductInterface
    {
        // Return Product object model that has to implement the interface ProductInterface
        return $this->productRepository->getByLanguage(new ProductId($productId), new ShopId($shopId), new
LanguageId($languageId));
    }

    public function getLocalizedProduct(int $productId, int $shopId): LocalizedProductInterface
    {
        // Return Product object model that has to implement the interface LocalizedProductInterface
        return $this->productRepository->get(new ProductId($productId), new ShopId($shopId));
    }
}
```

Doctrine example

```
namespace PrestaShopBundle\Model\Repository;

use PrestaShopBundle\Entity\Repository\ShopRepository as DoctrineShopRepository;

class ShopRepository implements ShopRepositoryInterface
{
    private $shopRepository;

    public __contruct(DoctrineShopRepository $shopRepository) {
        $this->shopRepository = $shopRepository;
    }

    public function getShop(int $shopId): ShopInterface
    {
        // Return Shop Doctrine entity that has to implement the interface ShopInterface
        return $this->shopRepository->findById($shopId);
    }
}
```

Legacy static method example

```
namespace PrestaShop\PrestaShop\Adapter\Security;

class Access
{
    public function getRoles(): array
    {
        $rolesFlattenArray = LegacyAccess::getRoles($employeeProfileId);

        // Transform flatten array into interfaces (implemented by DTO probaly)
        return array_map([$this, 'turnArrayToInterface'], $rolesFlattenArray);
    }

    private function turnArrayToInterface(array $data): RoleInterface
    {
        return new RoleDTO(...);
    }
}

namespace PrestaShopBundle\Model\Repository;

use PrestaShop\PrestaShop\Adapter\Security\Access as AdapterAccess;

class RoleRepository implements RoleRepositoryInterface
{
    private $access;

    public __contruct(AdapterAccess $access) {
        $this->access = $access;
    }
```

```
    /**
     * @return RoleInterface[]
     */
    public function getRoles(): array
    {
        return $this->access->getRoles();
    }
}
```

4. Namespaces and rules

Folder (from root)	Namespace	Content	Rules
`src/Core/Model`	`PrestaShop/PrestaShop/Core/Model`	All the model interfaces, maybe split by domain at least for large ones like `Product`, should also contain the DTOs if/when needed	Only allows usage of sc `DecimalNumber`, `ShopC` code from the `PrestaShop/PrestaSh` namespace
`src/Core/Model/Repository`	`PrestaShop/PrestaShop/Core/Model/Repository`	The repository interfaces (same could be split into sub-namespaces for large domains)	Only allows usage of sc `DecimalNumber`, `ShopC` code from the `PrestaShop/PrestaSh` namespace
`src/PrestashopBundle/Repository`	`PrestaShopBundle/Repository`	All the implementations of the interfaces are here, regardless of the implementation (DBAL, ObjectModel, Doctrine, ...) this allows being sure where to find the repository	No usage of legacy obj
`src/Adapter/{Model}/Repository`	`PrestaShop\PrestaShop\Adapter\{Model}\Repository`	Repositories based on `AbstractObjectModelRepository` based on `ObjectModel` classes, don't need to implement the model repository interface as it can implement it only partially, method based on static call should also be here (to avoid putting them in the bundle namespace)	I don't see any particula needed
`src/PrestashopBundle/Entity/Repository`	`PrestashopBundle/Entity/Repository`	Repositories based on Doctrine entities	No usage of legacy clas