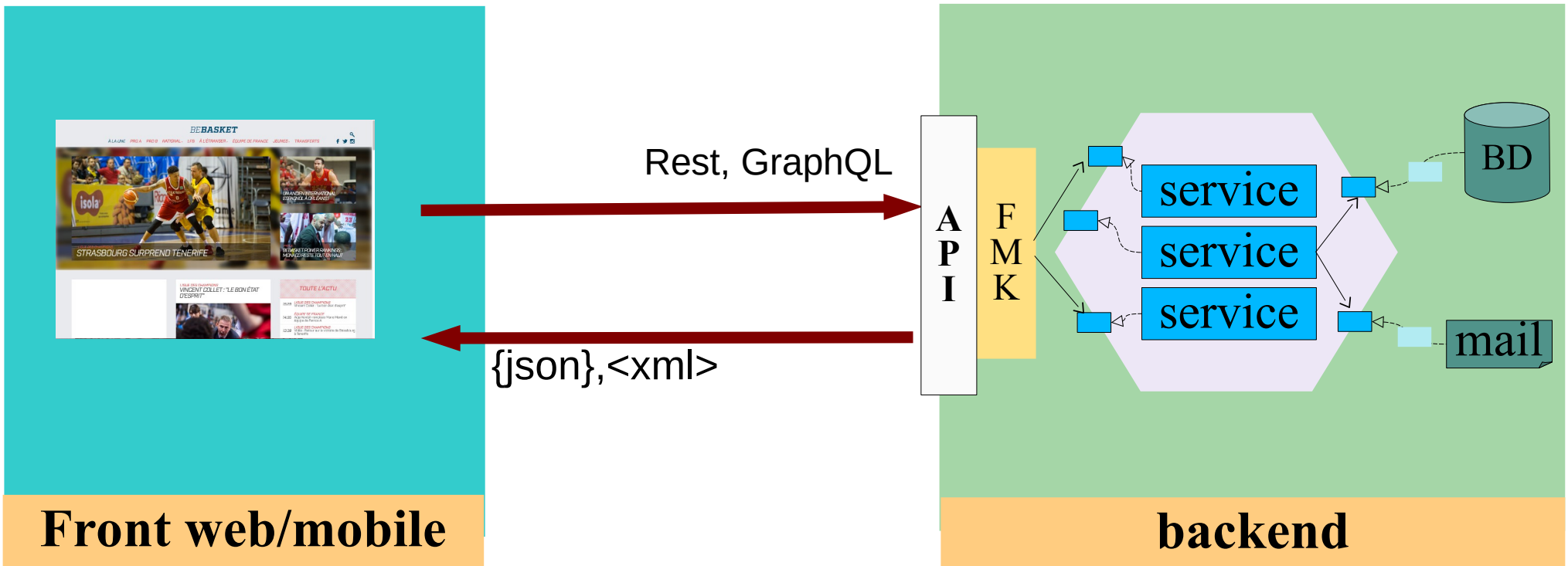
Compléments sur l’architecture d’un backend web

Contexte : application monopage (SPA)

- L'interface est générée et exécutée côté frontend

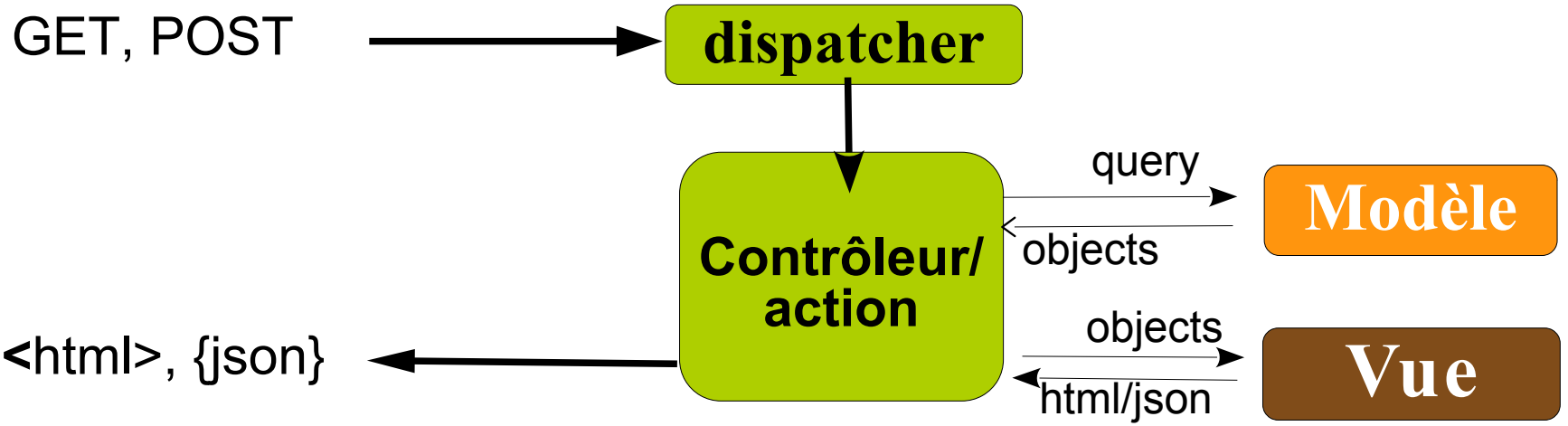


Intérêt

- Le frontend dépend uniquement de l’API
- Plusieurs frontend peuvent partager la même API
- Frontend et Backend peuvent être développés et testés séparément
- Le backend peut évoluer sans impacter le frontend tant qu’il respecte l’API
- Le frontend peut évoluer sans impacter le backend tant qu’il respecte l’API

Architecture du backend : MVC

- **Rappel** : Modèle-Vue-Contrôleur



- Intérêt :
  - séparation des préoccupations
  - structuration simple

Architecture du backend : MVC

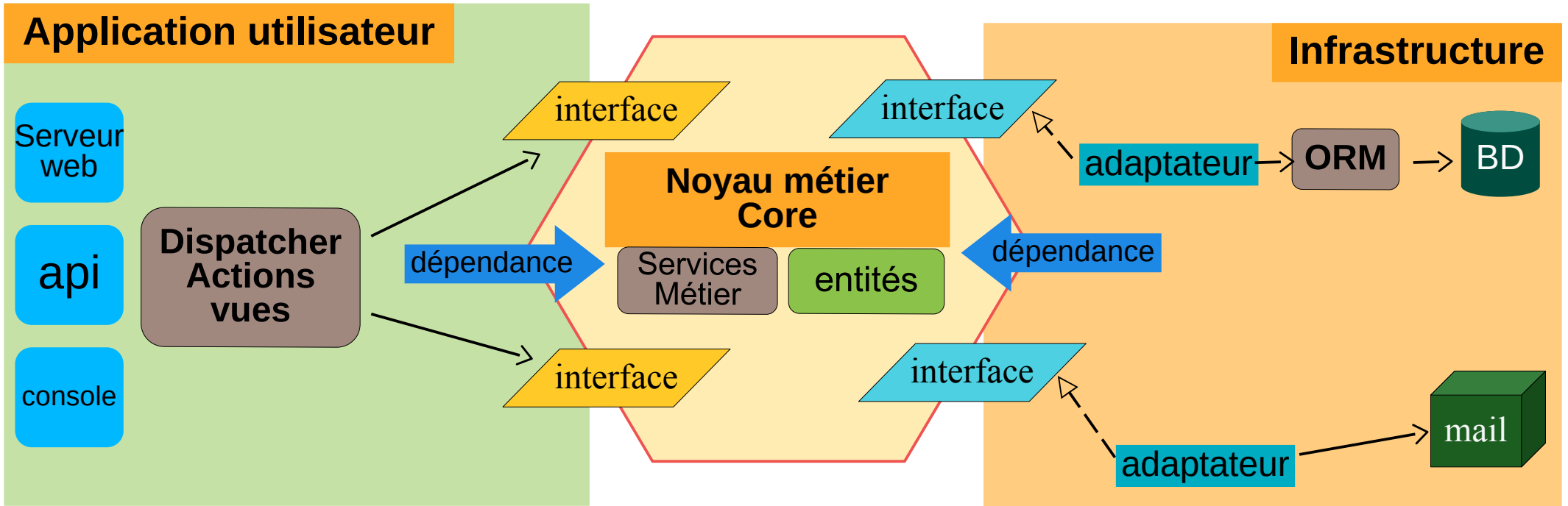
- En appliquant le modèle MVC, le métier de l'application (fonctionnalités, cas d'utilisation, processus, règles de gestion) est réalisé en partie dans les contrôleurs et en partie dans les modèles
  - ➔ **le métier est dépendant du framework de développement et de l'ORM**
- Les fonctionnalités métier au coeur de l'application
  - ➔ sont **impossibles à tester** individuellement, et en dehors du framework et de l'ORM
  - ➔ sont **non réutilisables** dans différentes applications
  - ➔ sont **difficiles à maintenir et à faire évoluer**
  - ➔ sont **difficiles à porter** sur une infrastructure différente

Architecture du backend : MVC

- Bien adapté pour des applications de petite taille et/ou à faible durée de vie
- Avec un domaine métier simple et réduit

Architecture Hexagonale pour le backend

- **Rappel :** application – noyau métier – infrastructure



- Le métier ne dépend de rien
- L'application et l'infrastructure dépendent du métier

Architecture hexagonale : intérêt

- Le noyau métier **ne dépend de rien** et donc :

- Il est **testable indépendamment**
  - notamment, pas besoin du framework web ou de requêtes HTTP pour le tester
  - pas besoin de base de données ou d'ORM pour le tester
- Il est **réutilisable** pour construire différentes applications
  - appli html, API json web et/ou mobile, console, standalone
- Il est **portable** d'une infrastructure à une autre
  - BD interchangeable, messagerie interchangeable ...

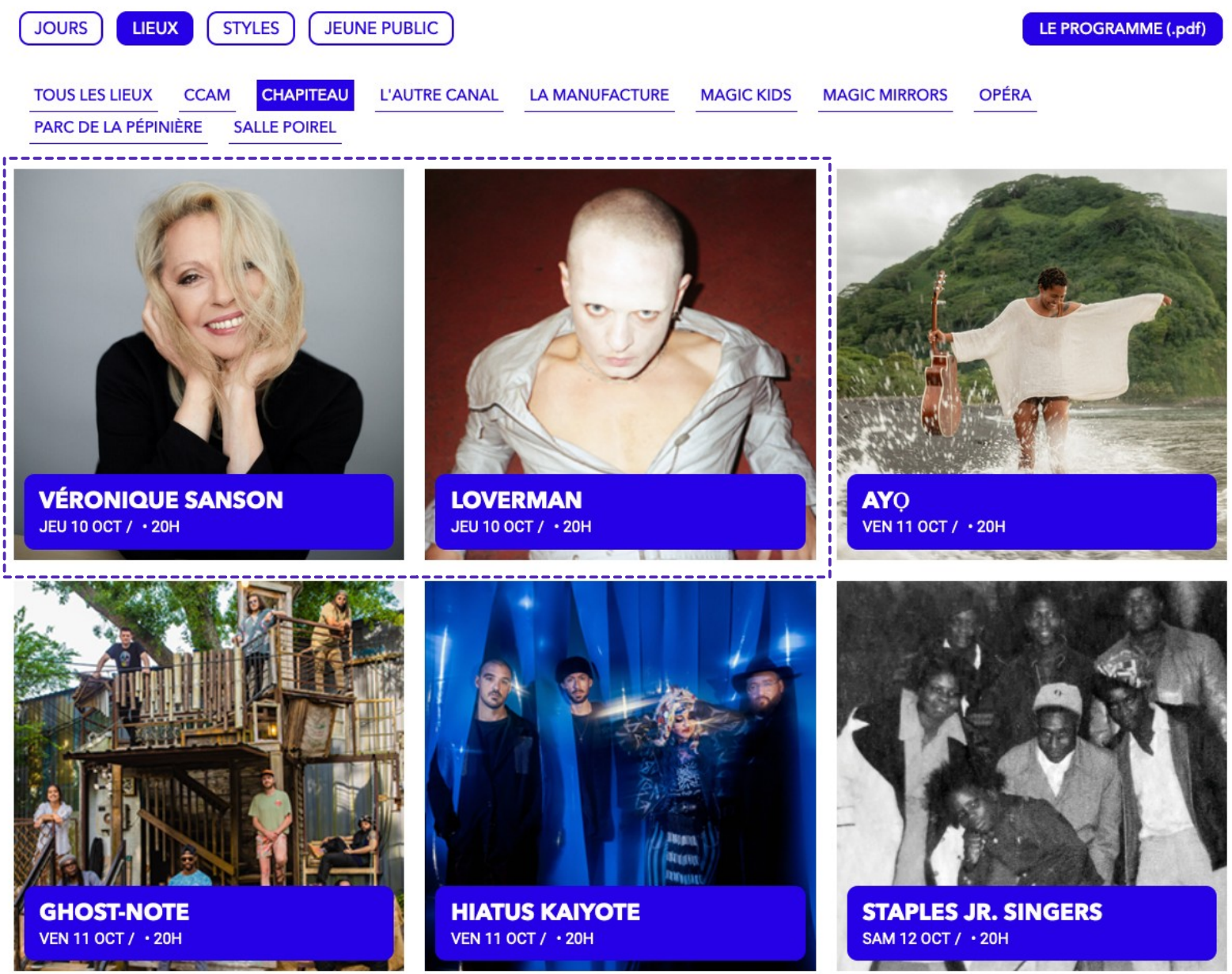
Condition de réalisation

- Les entités et services métiers n'utilisent
  - aucune classe ou interface de la couche application et donc du framework web (y compris requêtes/réponses, exceptions et erreurs)
  - aucune classe ou interface de l'infrastructure, et notamment n'utilisent pas l'ORM

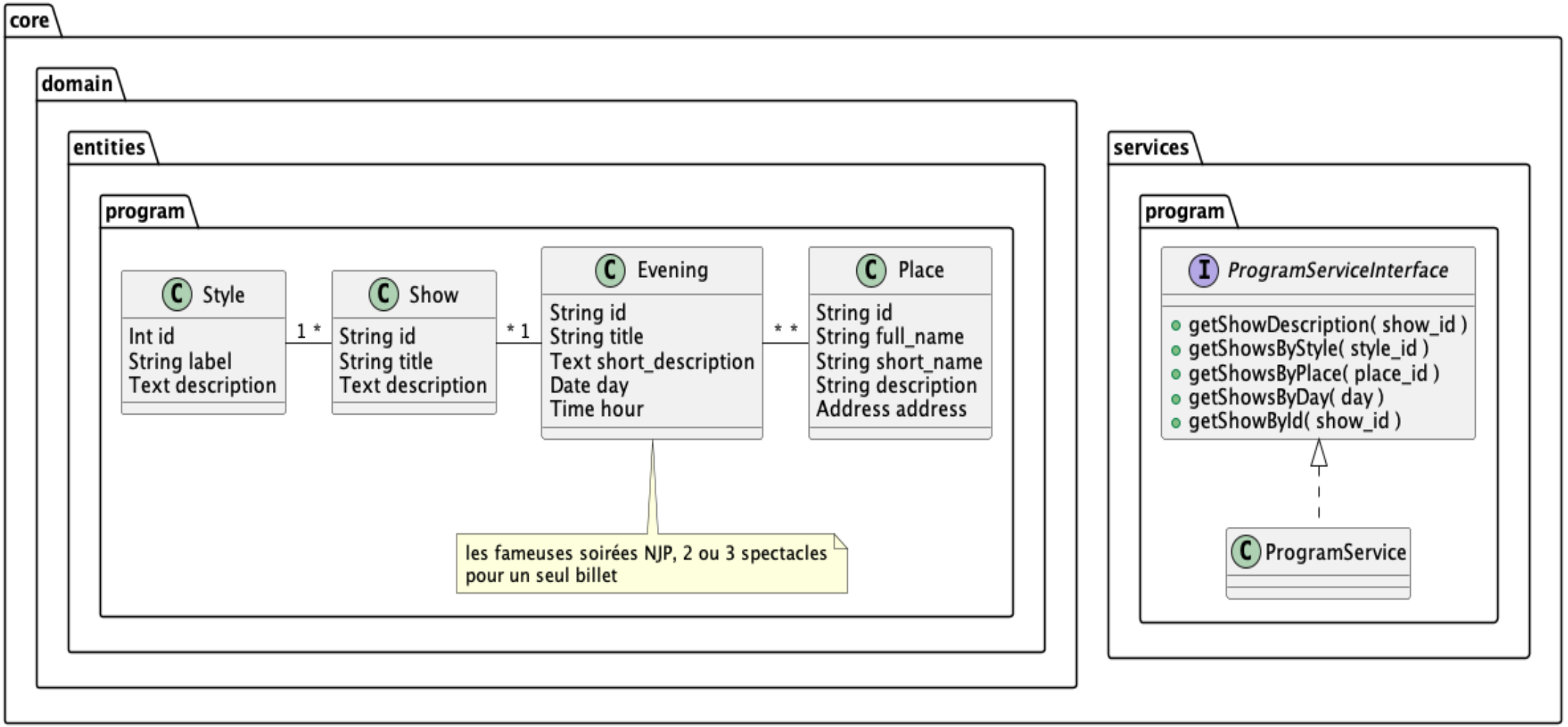
Exemple : la programmation du NJP

1 soirée

+ spectacles

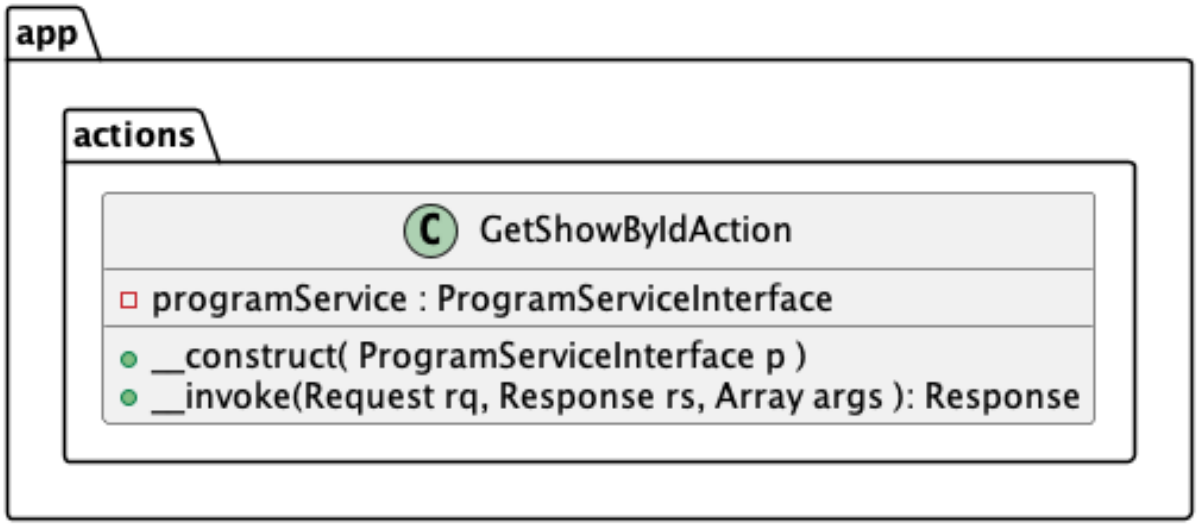


La programmation du NJP



Dépendances côté application

- La dépendance est naturellement orientée application → métier
  - les actions utilisent les services métiers

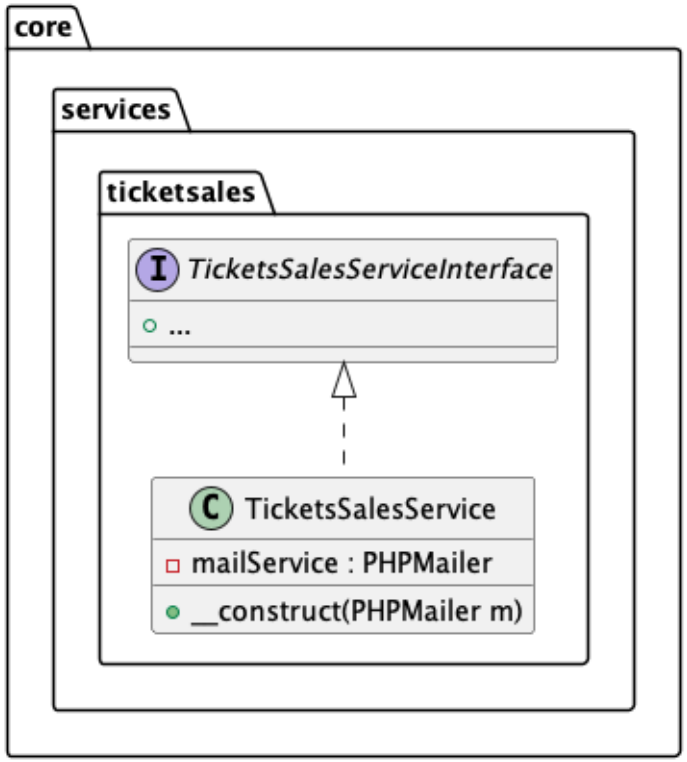


- Les actions ne connaissent que les interfaces exposées par le noyau métier : cela rend les services concrets interchangeables

Dépendances côté infrastructure

- Exemple : le service de vente de tickets NJP souhaite envoyer des mails de confirmation aux acheteurs
  - Service de vente de tickets : noyau métier
  - Service d'envoi de mail : infrastructure
- Si on ne fait pas attention, on crée une dépendance dans le sens service de vente → service mail

➔ Le service métier dépend de l'infrastructure :-(



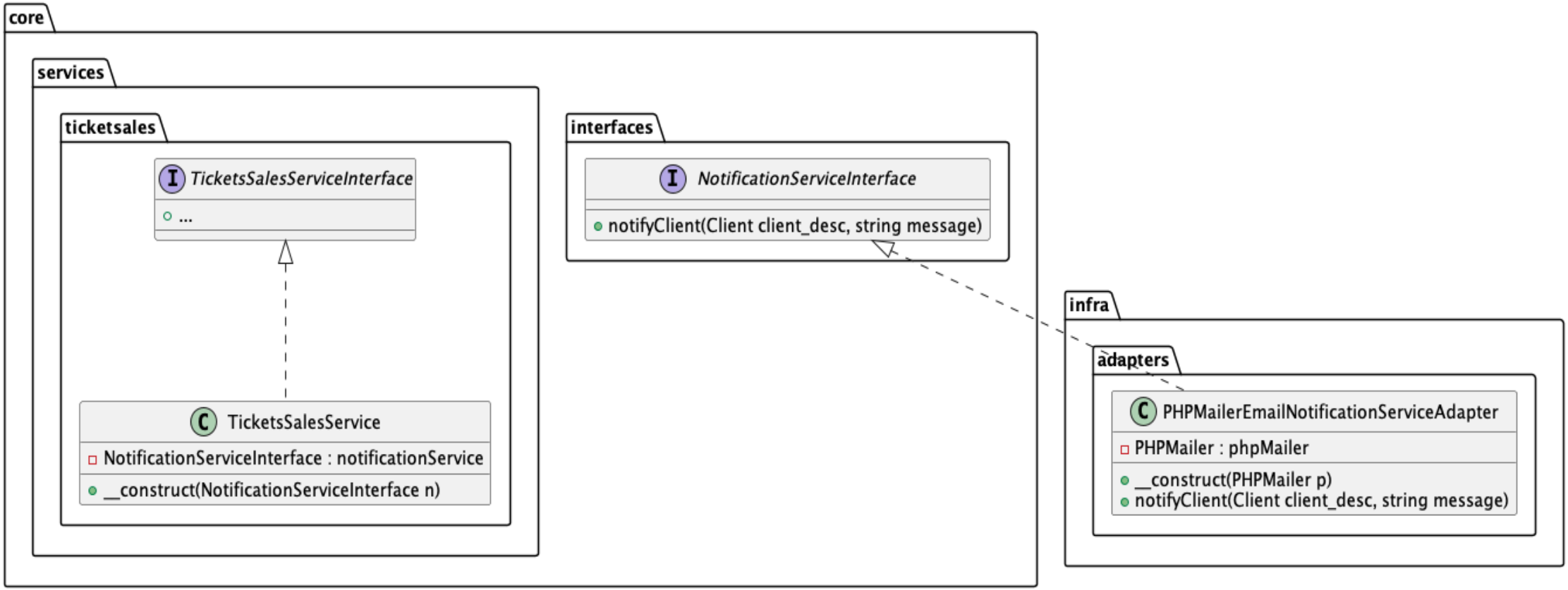
Rappel de conception objet

- Faites appel à vos souvenirs de conception objet 1ère et 2ème année

- **Comment peut-on faire pour inverser cette dépendance ?**

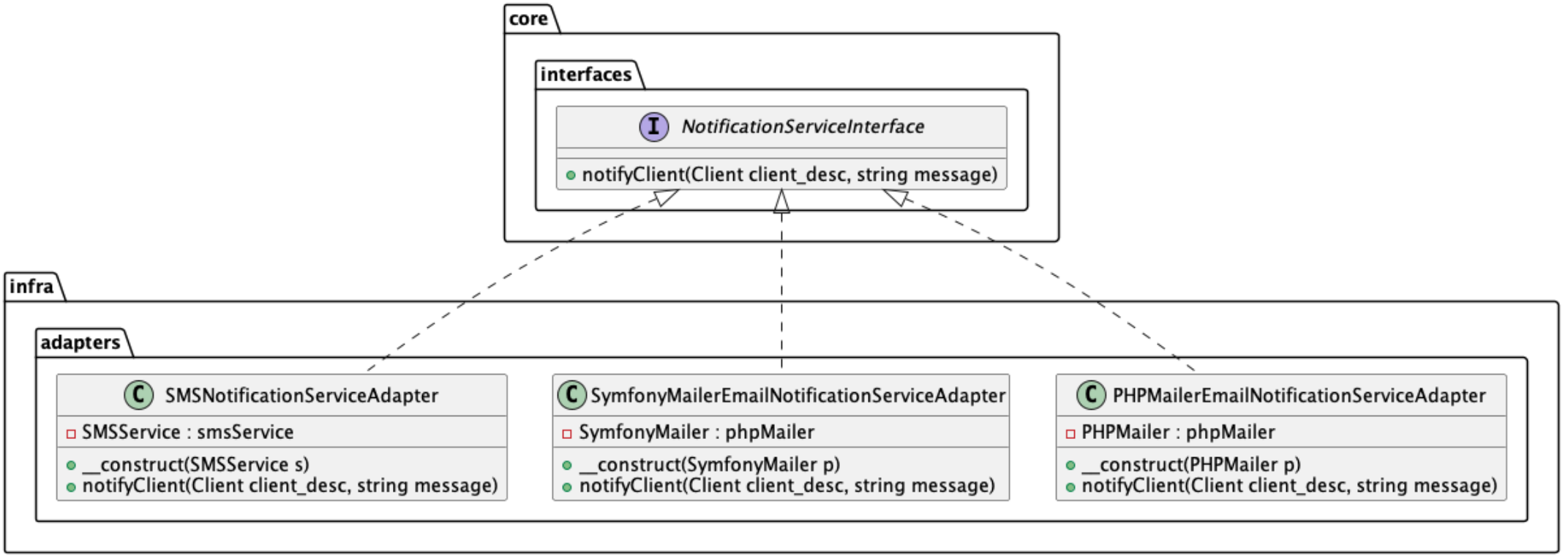
réponse

- En ajoutant une interface et un Adaptateur
  - l'interface est définie par le métier
  - l'adaptateur est implanté par l'infra
- La dépendance est inversée : l'infra implante une interface métier



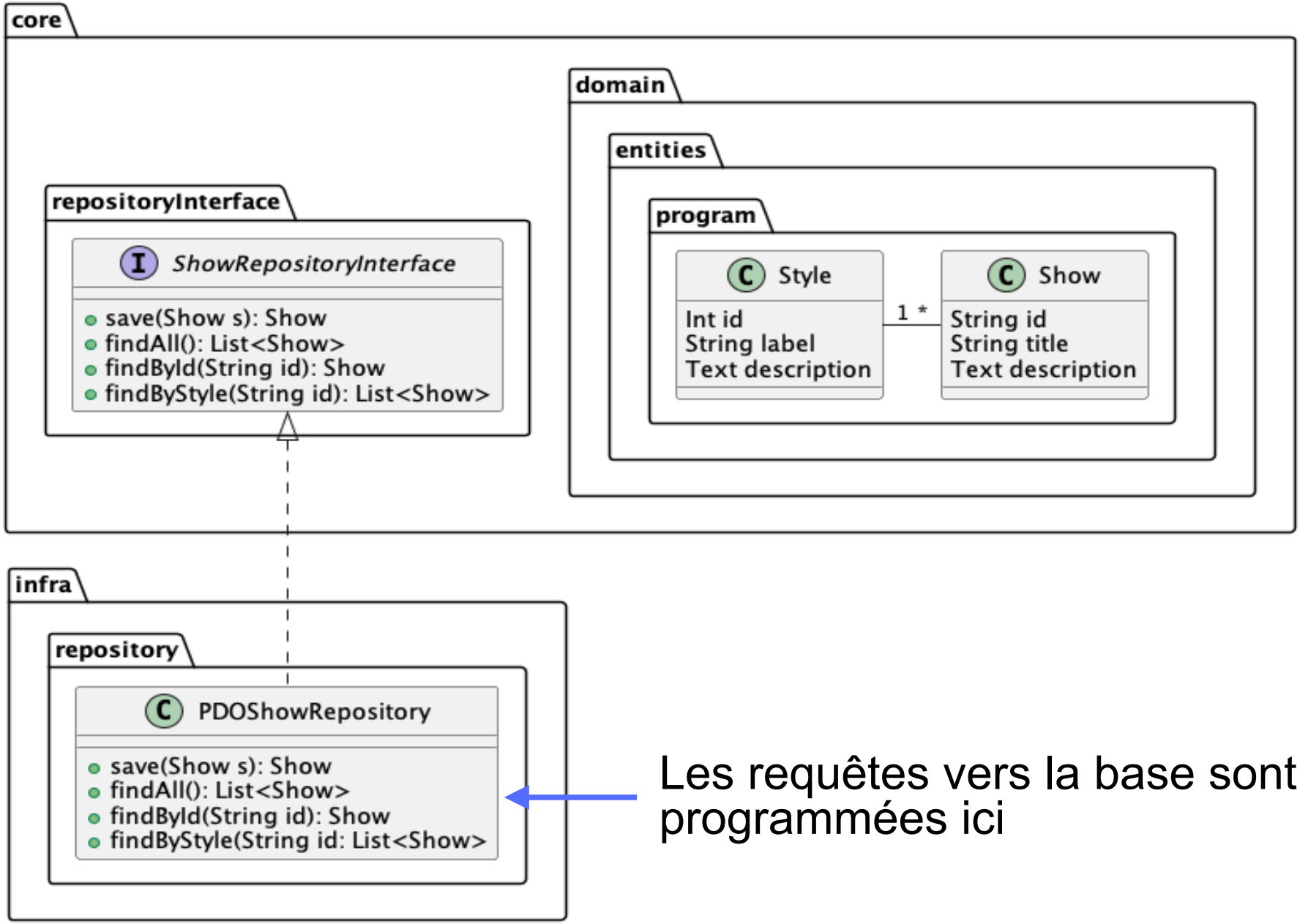
Intérêt

- L’infrastructure devient remplaçable :



Inversion de la dépendance dans le cas de la base de données

- On applique le même principe : interface métier + adaptateur dans l'infrastructure
- L'adaptateur est un objet intermédiaire entre les entités métiers et la persistance, capable de
  - faire des requêtes dans la base en retournant des entités métier
  - recevoir des entités et les insérer/sauvegarder dans la base
- L'adaptateur implante une interface définie par le métier en fonction de ses besoins
- Ce principe est connu sous le terme de **pattern Repository**



- L'infrastructure implante une interface métier : la dépendance est inversée

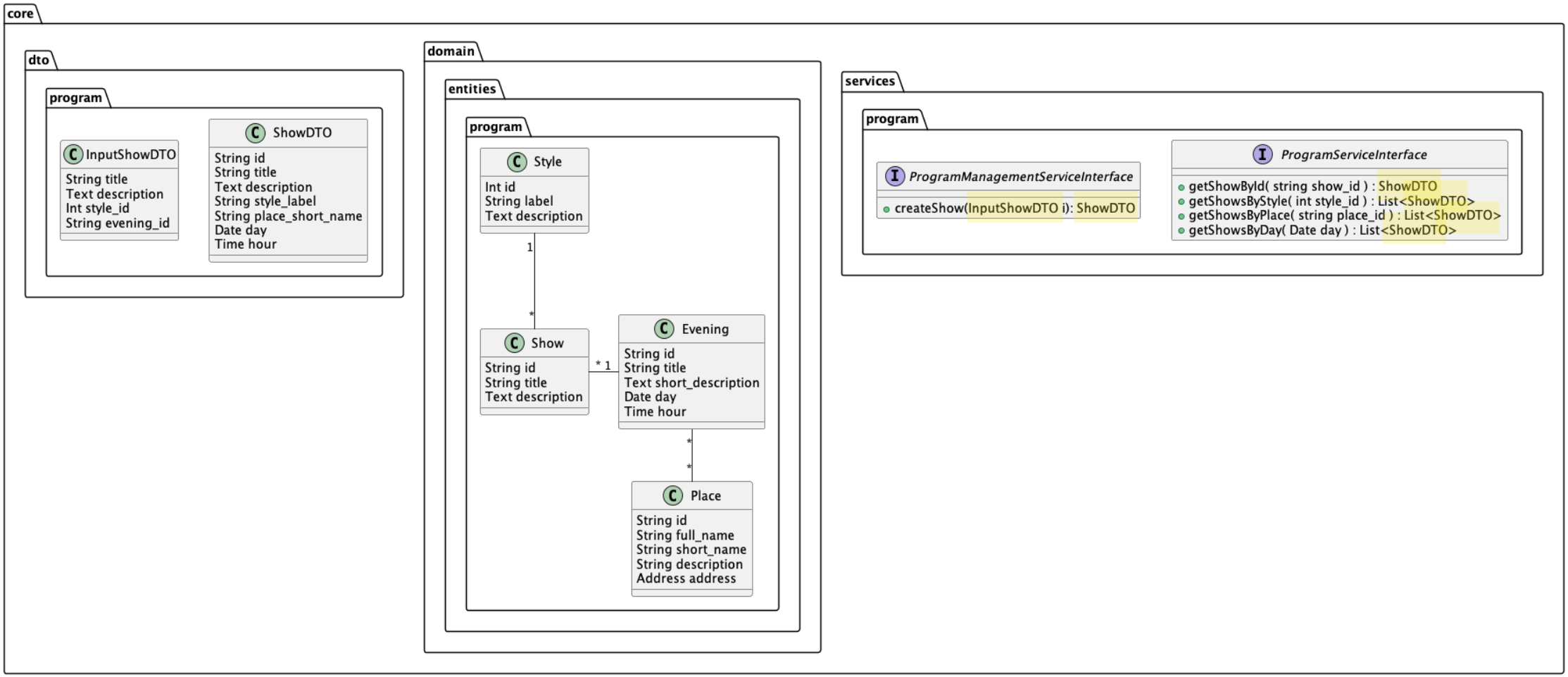
Échanges de données entre application et services métier

- Les services métier sont appelés par la couche application ou par d'autres services au travers de leur interface
- Les méthodes de ces interfaces reçoivent des données et retournent des données
- On ne doit pas utiliser les entités du domaine métier pour limiter le couplage
- On utilise le pattern DTO : Data Transfer Object

La notion de DTO

- Un DTO est un objet neutre servant uniquement à transférer des données d'un composant à un autre
  - des actions vers les services : par exemple les données issues d'un formulaire
  - des services vers les actions : par exemple le résultat d'une fonctionnalité appelée par l'action
  - d'un service métier vers un autre service métier
- Les DTO sont utilisés pour typer les paramètres et résultats des méthodes listées dans les interfaces métier des services ; ils sont définis par le métier
- On peut créer autant de DTO que nécessaire
  - DTO en entrée, en sortie, partiels ...

Exemple : show DTO



- Un DTO en entrée pour créer un spectacle
- Un DTO en sortie pour décrire un spectacle

Validation de données

- Les données reçues dans l'interface de l'application pour être utilisées par les services métier et stockées de manière persistante doivent être vérifiées pour s'assurer qu'elles sont conformes
  - pour éviter de créer des erreurs d'exécution
  - pour garantir au mieux la sécurité
  - pour assurer la validité des fonctionnalités et des informations stockées
- Cela concerne toutes les données : paramètres de recherches, données pour création d'entités
  - attention particulière aux données issues de formulaires destinées à être stockées en base de données

Que vérifier ?

- **Présence** : les données transmises sont-elles complètes ?
- **Structure et types** : les données sont-elles du type attendu, la structure est-elle correcte ?
- Filtrage et nettoyage pour la **sécurité** : pour éviter les injections
- **Valeurs** : les valeurs transmises sont-elles conformes aux contraintes d'intégrité et règles de gestion du métier ?
  - exemple : date de livraison > date de commande

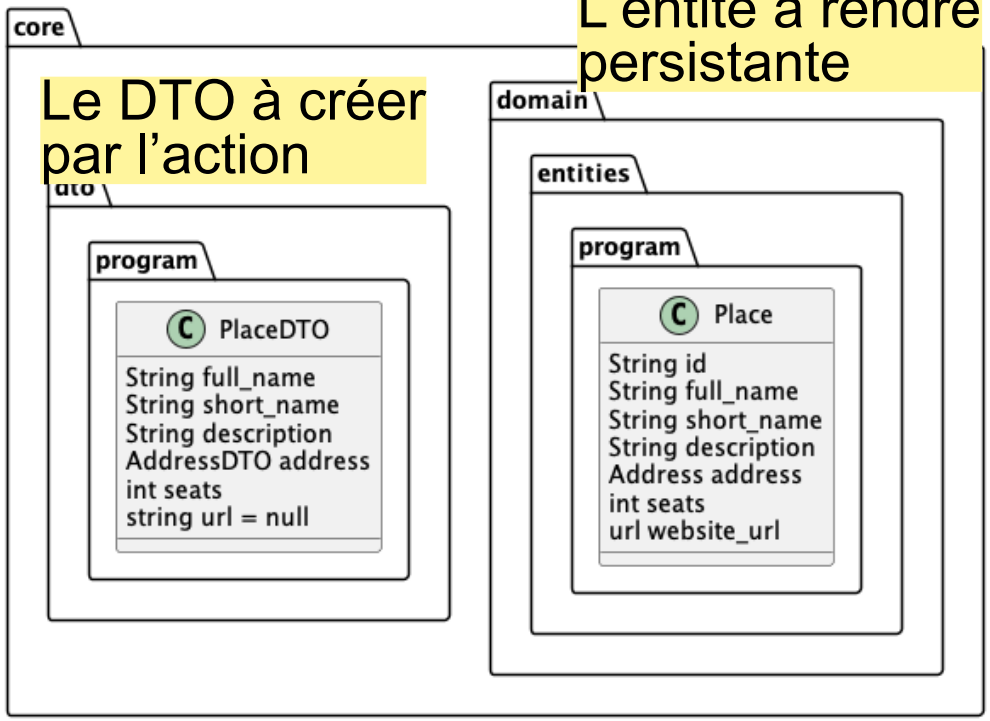
Exemple : création d'un lieu de spectacle NJP

Les données reçues du formulaire dans l'action

app

actions

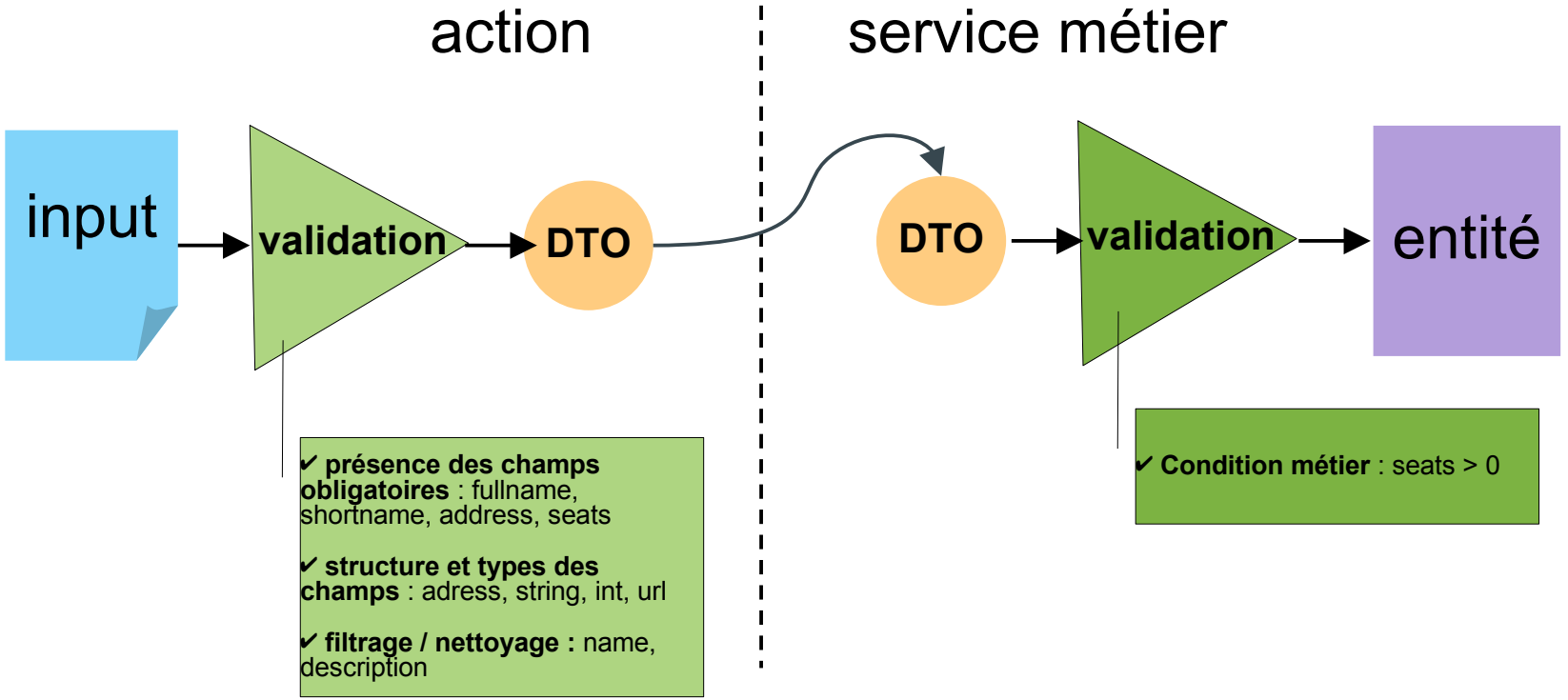
PlaceInput		
input_full_name	Le Chapiteau NJP	
input_short_name	Chapiteau	
input_description	Le Chapiteau, lieu mythique du NJP et centre névralgique du festival	
input_address	input_street	Parc de la Pépinière
	input_postal_code	54000
	input_city	Nancy
	input_country	France
input_seats	2500	
input_website_url	https://www.nancyjazzpulsations.com/le-chapiteau/	



- Valider la présence des champs obligatoires : full name, short name, description, address, seats
- Valider la structure de l'adresse
- Valider les types : string, int, url
- Filtrer / nettoyer pour éviter les injections
- Valider la condition métier : seats > 0

Où valider ?

- Dans la couche application, avant la création du DTO :
  - présence, structure, types, filtrage/nettoyage
- Dans les services métiers :
  - conditions métier : contraintes et règles liées à la gestion



Comment valider ?

- Fastidieux et répétitif

➔ utiliser une librairie spécialisée

- **respect/validation** : librairie de validation de données en php
  - des validateurs : conditions à vérifier (type, format, valeurs ...)
  - des fonctions de validation : appliquent les validateurs à des données [structurées] et génèrent des messages d'erreurs
- Documentation :

  https://respect-validation.readthedocs.io/en/2.3/02-feature-guide/

  https://respect-validation.readthedocs.io/en/2.3/08-list-of-rules-by-category/

Exemple : validateur des données de création d'un lieu (tableau)

```php
use Respect\Validation\Validator;
use Slim\Exception\HttpBadRequestException;

class createPlaceAction
{
  public function __invoke(ServerRequestInterface $rq,
                           ResponseInterface $rs,
                           array $args): ResponseInterface
  {

    $data = $rq->getParsedBody() ?? null;

    $placeInputValidator =
      Validator::key('full_name', Validator::stringType()->notEmpty())
        ->key('short_name', Validator::stringType()->notEmpty())
        ->key('description', Validator::stringType()->notEmpty())
        ->key('address', Validator::arrayType()
            ->key('street', Validator::stringType()->notEmpty())
            ->key('city', Validator::stringType()->notEmpty())
            ->key('zip', Validator::stringType()->notEmpty())
            ->key('country', Validator::stringType()->notEmpty())
          )
        ->key('seats', Validator::intType())
        ->key('url', Validator::optional(
                                  Validator::stringType()->notEmpty()));



  /* … */
  }
}
```

Exemple : validation des données dans l'action

```php
use Respect\Validation\Validator;
use Slim\Exception\HttpBadRequestException;

class createPlaceAction
{
  public function __invoke(ServerRequestInterface $rq,
                           ResponseInterface $rs,
                           array $args): ResponseInterface
  {

     $data = $rq->getParsedBody() ?? null;

     $placeInputValidator = … ;
     try {
          $placeInputValidator->assert($data);
     } catch (\Respect\Validation\Exceptions\NestedValidationException $e) {
            throw new HttpBadRequestException($rq, $e->getMessages());
     }

     if ((filter_var($data['description'],
                     FILTER_SANITIZE_FULL_SPECIAL_CHARS)!== $data['description'] ||
          filter_var($data['full_name'],
                     FILTER_SANITIZE_FULL_SPECIAL_CHARS) !== $data['full_name']    ||
          filter_var($data['short_name'],
                     FILTER_SANITIZE_FULL_SPECIAL_CHARS) !== $data['short_name'])
     ) {
          throw new HttpBadRequestException($rq, 'Bad data format');
     }

     $placeDTO = new PlaceDTO($data);

  /* … */
  }
}
```

Validations métier

- Bonne pratique : le service métier injecte un validateur dans le DTO puis déclenche la validation

```php
abstract class DTO
{
  protected Validatable $business_validator;

  public function setBusinessValidator(Validatable $validator): void
  {
        $this->business_validator = $validator;
  }
  public function validate(): void {
        $this->business_validator->assert($this);
  }
  public function toJSON(): string {
        return json_encode($this, JSON_PRETTY_PRINT);
  }
}
```

Exemple : validation métier dans le service

```php
use Respect\Validation\Validator;

class ProgramManagementService
{
  public function createPlace(PlaceDTO $placeDTO): void {
   $placeValidator = Validator::attribute('seats', Validator::intVal()->positive());

   $placeDTO->setBusinessValidator( $placeValidator);

   try {
        $placeDTO->validate();
   } catch (\Respect\Validation\Exceptions\NestedValidationException $e) {
       throw new ServiceDataException('Invalid place data: ' . $e->getMessages());
   }

   // save place to database
   // ...
 }
}
```