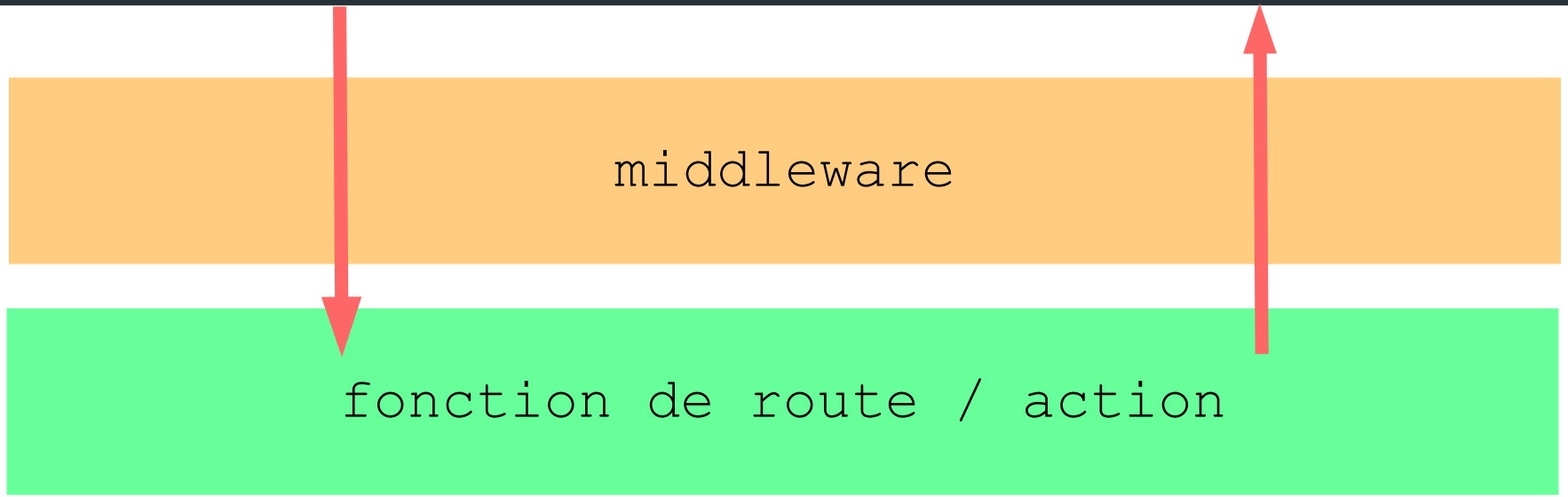
# Les Middleware, utilisation dans Slim

- **Définition :** un *middleware* est une fonction invoquée lors de l'activation d'une route, qui peut modifier la requête ou la réponse courante
  - ***avant*** l'exécution de la fonction de route, puis
  - ***après*** l'exécution de la fonction de route

```
$app->get('/hello/', function() { … })
    ->add( function() { … } ) ;
```

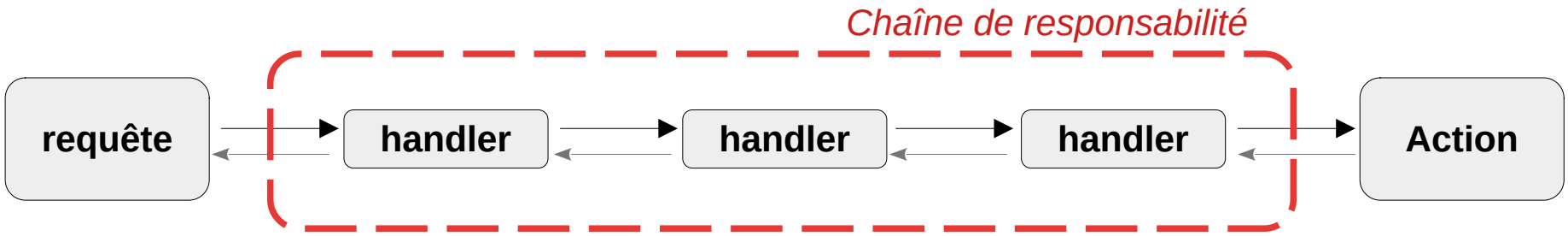


# Intérêt, utilisation

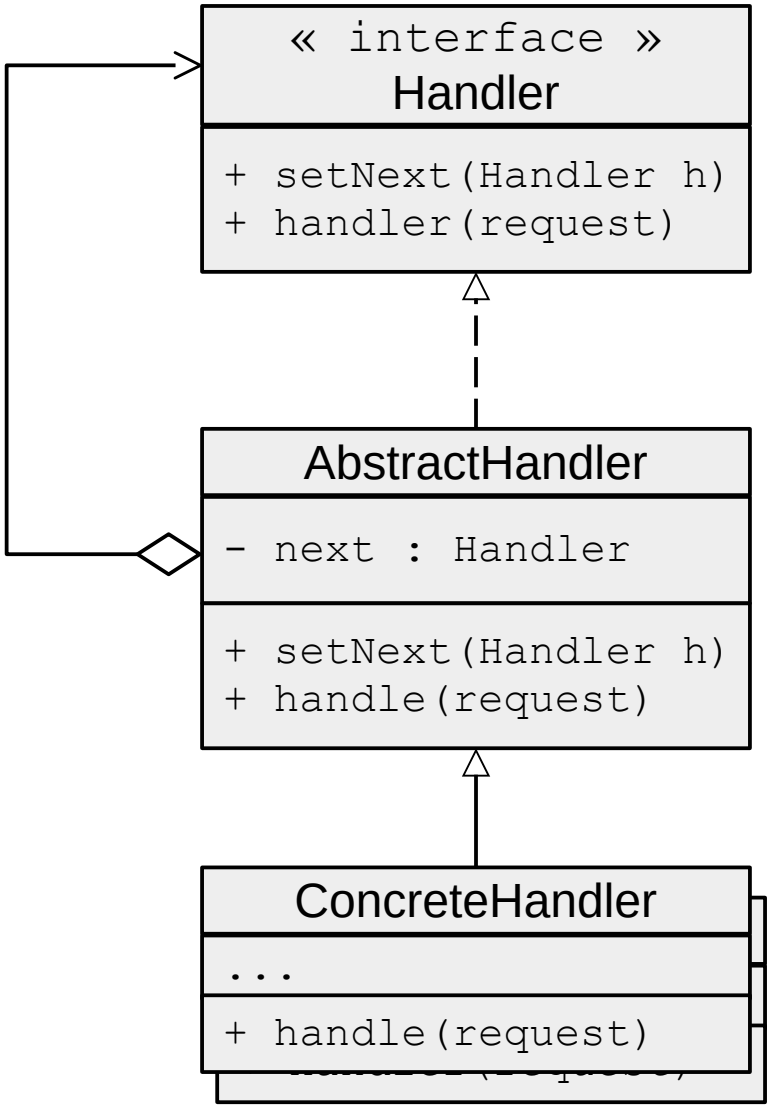
- Rendre le framework/application extensible en ajoutant des fonctionnalités
- Factoriser du code commun à plusieurs routes
- Exemples :
  - contrôle de token et d’authentification
  - Contrôle d’autorisations
  - ajout de headers : content-type, header CORS
  - validation de données, création de DTO
  - logging d'activité

# Principe

- Basé sur le patron de conception "Chaine de responsabilité"
  - un message ou une requête est traité successivement par une série de *handler* indépendants les uns des autres
  - chaque *handler* traite (ou non) la requête reçue et la transmet à son successeur



# Structure de la chaine de responsabilité



# Middleware Slim

- 1 **middleware** = 1 *callable* (fonction, méthode, __invoke)
- recevant 2 **paramètres** :
  - 1 requête `\Psr\Http\Message\ServerRequestInterface`
  - le handler suivant dans la liste, `\Psr\Http\Server\RequestHandlerInterface`
- retournant 1 **réponse** `\Psr\Http\Message\ResponseInterface`

```php
function( Request $rq, RequestHandlerInterface $next): Response
{
    // actions avant la fonction de route

    // appel du handler suivant ou de la fonction de route
    $rs = $next->handle($rq);

    // actions après la fonction de route

    return $rs ;
}
```

# Enregistrement de middleware

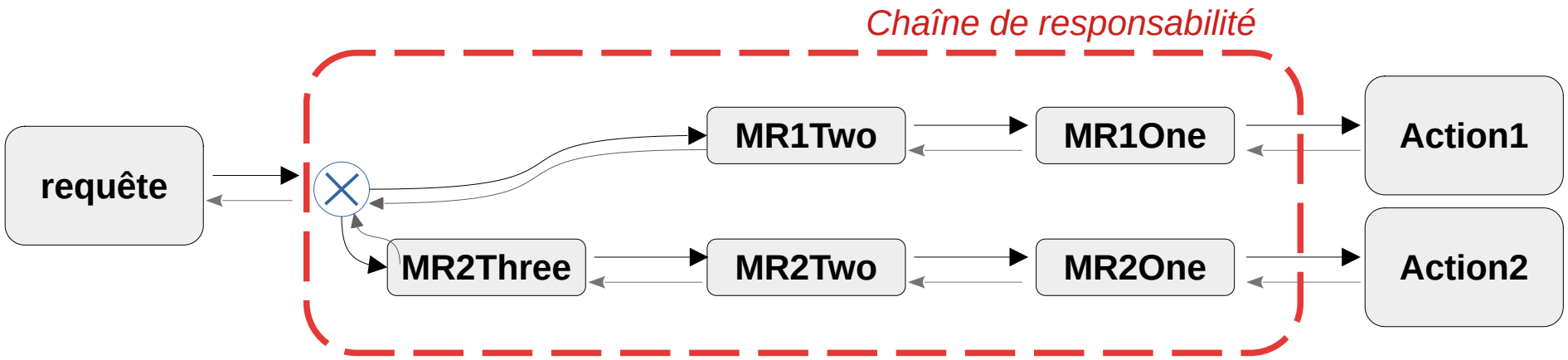
- Un middleware peut être enregistré :
  - pour l'ensemble de l'application : il est utilisé lors de l'activation de toutes les routes déclarées
  - pour une route particulière : il est utilisé lorsque cette route est activée

```php
$app = AppFactory::create();

$app->add(new \app\middlewares\MiddleApp());

$app->get('/mid', function(){ ... })
    ->add(new \m\MiddleOne())
    ->add(new \m\MiddleTwo());
```

# Enregistrement de middlewares

- La méthode add() empile le middleware reçu
- Elle gère une pile par route

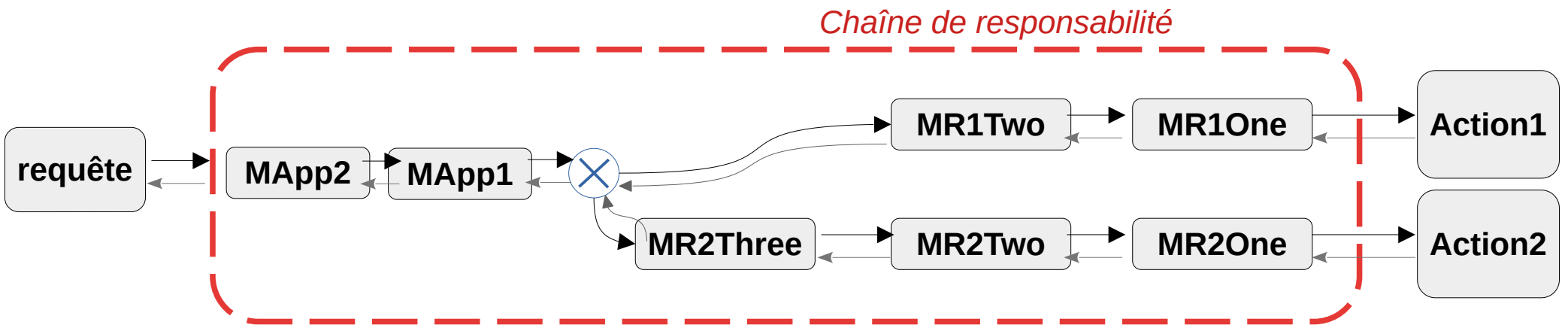
```php
$app->get('/route1', Action1::class)
    ->add(new \m\MR1One())
    ->add(new \m\MR1Two()) ;
$app->get('/route2', Action2::class)
    ->add(new \m\MR2One())
    ->add(new \m\MR2Two())
    ->add(new \m\MR2Three()) ;
```



# Enregistrement des middlewares

- Les middlewares d'application sont enregistrés dans une pile partagée placée devant les piles de routes

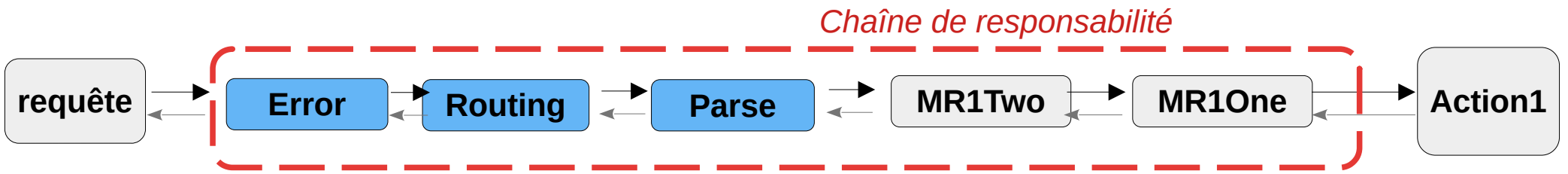
```php
$app->add(new \m\MApp1()) ;
$app->get('/route1', Action1::class)
    ->add(new \m\MR1One())
    ->add(new \m\MR1Two()) ;
$app->get('/route2', Action2::class)
    ->add(new \m\MR2One())
    ->add(new \m\MR2Two())
     ->add(new \m\MR2Three()) ; ;
$app->add(new \m\MApp2()) ;
```



# Slim et les middleware

- Slim utilise de façon systématique les middleware pour la réalisation de plusieurs fonctionnalités de base (erreurs, routage, parse du body), et fournit des middleware par défaut pour ces fonctionnalités :

```php
$app = AppFactory::create();

// middleware pour parser le body
$app->addBodyParsingMiddleware();
// le routage est réalisé par un middleware !
$app->addRoutingMiddleware();
// et la gestion des erreurs aussi !!
$app->addErrorMiddleware(true, false, false);
```

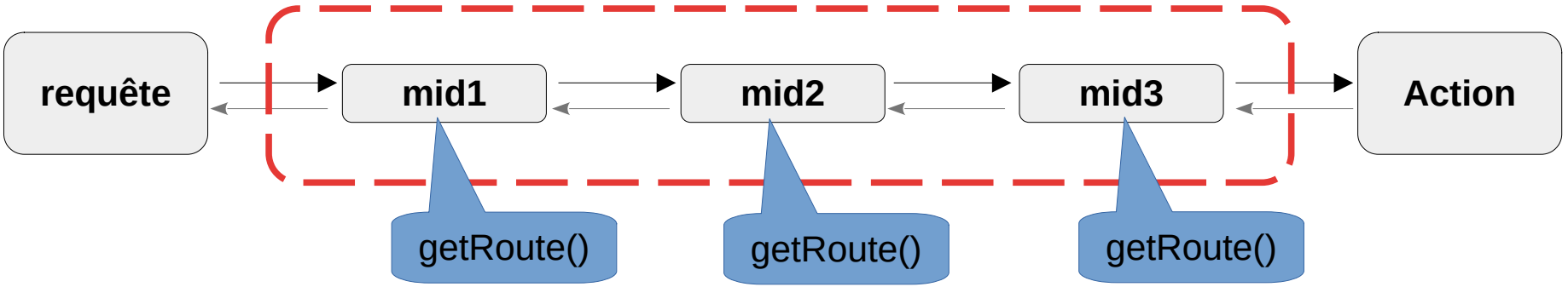



- Attention à l'ordre : le middleware d'erreur agit uniquement sur les middlewares placés en dessous dans la pile

# Accéder à la route courante dans un middleware

- La route courante s'obtient à partir d'un RouteContext instancié à partir de la requête courante
- Permet dans chaque middleware d'accéder aux caractéristiques de la route : nom, méthode http, arguments

```php
function( Request $rq,  RequestHandler $next) :Response {

    $route = RouteContext::fromRequest($rq)->getRoute() ;
    $name    = $route->getName() ;
    $methods = $route->getMethods() ;
    $id = $route->getArguments()['id'] ;
}
```



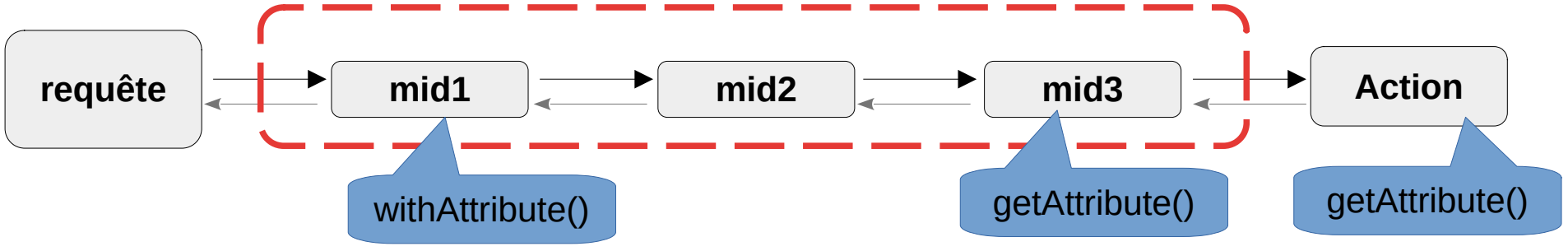
# Transmission de valeurs

- 1 middleware peut transmettre 1 valeur aux middlewares suivants ou à la fonction de route en ajoutant 1 attribut à la requête en cours
- Positionner la valeur dans le middleware :

```
$request = $request->withAttribute( 'yo' , 73 ) ;
```

- Récupérer la valeur dans le middleware suivant ou dans la fonction de route :

```
$bar = $request->getAttribute( 'yo'  ) ;
```



# Exemple : validation de clé d'api

- Middleware agissant sur la requête avant exécution de la fonction de route :
  - présence et validité d'une clé d'api : vérifier que l'url contient un paramètre 'apikey' avec une valeurs valide compte tenu de la ressource accédée

    `https://myapp.com/map/`**`5rrfse?apikey=345FA5B3D`**

```
$app->get('/tests/{id}', GetMapByIdAction::class

)->setName('map')
 ->add(new \app\middlewares\checkApiKey());
```

- Le middleware vérifie la présence d'une clé, contrôle sa validité pour la route concernée, et transmet sa valeur

```php
class ChekApiKey {

  public function __invoke(
      ServerRequestInterface $request,
      RequestHandlerInterface $next    ): ResponseInterface {

    $ressourceId = RouteContext::fromRequest($request)
                        ->getRoute()
                        ->getArguments()['id'] ;

    $key = $request->getQueryParams()['apikey'] ?? null;
    if (is_null($key))
      throw new HttpBadRequestException($request,'missing api key');

    try {
       $this->apiKeyService->checkApiKeyValidity($ressourceId, $key);
      } catch (ApiKeyServiceInvalidKeyException $e) {
          throw new HttpForbiddenException($request,'invalid api key');
    }
    $request = $request->withAttribute('apikey', $key);
    $response = $next->handle($request);
    return $response;
  }

}
```

# Exemple 2 : ajout de headers réponse

- Middleware agissant sur la réponse après exécution de la fonction de route :
  - ajout de headers dans la réponse

```php
$app->get('/tests/{id}', GetMapByIdAction::class

)->setName('map')
 ->add(new \app\middlewares\ChekApiKey())
 ->add(new \app\middlewares\AddHeaders());
```

```php
class AddHearders {

  public function __invoke(
   ServerRequestInterface $request,
   RequestHandlerInterface $next): ResponseInterface {

    $response = $next->handle($request);

    return $response->withHeader('Content-Language', 'fr-FR')
                    ->withHeader('Cache-Control', 'max-age='. 60*60*2);
    }}
```

# Exemple 3 : validation de données et création de DTO

- Créer un DTO pour transmettre des données bien formées au services métier
- Pour créer le DTO, il faut valider les données reçues dans la requête :
  - présence, structure et  type des données dans le body
- Il faut également sanitizer ces données pour prévenir les injections

- Pour la validation : utiliser une librairie, par exemple Respect/Validation

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

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

class CreerLieuValidationMiddleware
{
  public function __invoke(ServerRequestInterface $rq,
                           RequestHandlerInterface $next): ResponseInterface
  {

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

    try {
     v::key('nom', v::stringType()->notEmpty())
        ->key('nom_court', v::stringType()->notEmpty())
        ->key('description', v::stringType()->notEmpty())
        ->key('addresse', v::arrayType()
            ->key('rue', v::stringType()->notEmpty())
            ->key('ville', v::stringType()->notEmpty())
            ->key('code', v::stringType()->notEmpty())
          )
        ->key('places', v::intType())
        ->key('url', v::optional(v::stringType()->notEmpty()))
     ->assert($data);
   } catch (NestedValidationException $e) {
     throw
       new HttpBadRequestException($request, "Invalid data: " . $e->getFullMessage(),
$e);

    }



  /* … */
  }
}
```

# Exemple : sanitizer puis créer le DTO

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

class CreerLieuValidationMiddleware
{
  public function __invoke(ServerRequestInterface $rq,
                           RequestHandlerInterface $next): ResponseInterface
  {

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

    try {
      ...->assert($data);
  } catch (NestedValidationException $e) {
     throw new HttpBadRequestException($rq, "bad data: " . $e->getFullMessage(), $e);
  }
     if ((filter_var($data['description'],
                        FILTER_SANITIZE_FULL_SPECIAL_CHARS)!== $data['description'] ||
          filter_var($data['nom'],
                        FILTER_SANITIZE_FULL_SPECIAL_CHARS) !== $data['nom']    ||
          filter_var($data['nom_courte'],
                        FILTER_SANITIZE_FULL_SPECIAL_CHARS) !== $data['nom_court'])
      ) {
            throw new HttpBadRequestException($rq, 'Bad data format');
      }

      $lieuDTO = new LieuDTO($data);

       $request = $rq->withAttribute('lieu_dto', $lieuDTO);

       return $next->handle($request);


  }
}
```

# PSR-15

- PSR-15 standardise les middleware
- Slim peut utiliser à priori tous les middleware PSR-15
  - il existe des collections de middleware PSR-15 pour différents usages