plain-to-class Documentation

Release latest

Aug 01, 2023

CONTENTS

1 Installation **3**
- 1.1 Composer Install . . . 3

2 Usage **5**
- 2.1 Collection . . . 6
- 2.2 Anonymous array . . . 6
- 2.3 Writing style . . . 8
- 2.4 Alias . . . 8
- 2.5 Custom setter . . . 9
- 2.6 After Transform . . . 9
- 2.7 Custom transform . . . 9
- 2.8 Cache . . . 10

This library will allow you to easily convert any data set into the object you need. You are not required to change the structure of classes, inherit them from external modules, etc. No dancing with tambourines - just data and the right class.

It is considered good practice to write code independent of third-party packages and frameworks. The code is divided into services, domain zones, various layers, etc.

To transfer data between layers, the **DataTransfer Object** (DTO) template is usually used. A DTO is an object that is used to encapsulate data and send it from one application subsystem to another.

Thus, services/methods work with a specific object and the data necessary for it. At the same time, it does not matter where this data was obtained from, it can be an http request, a database, a file, etc.

Accordingly, each time the service is called, we need to initialize this DTO. But it is not effective to compare data manually each time, and it affects the readability of the code, especially if the object is complex.

This is where this package comes to the rescue, which takes care of all the work with mapping and initialization of the necessary DTO.

CHAPTER

ONE

INSTALLATION

The package can be installed via composer:

1.1 Composer Install

```
$ composer require yzen.dev/plain-to-class
```

CHAPTER

TWO

USAGE

Common use case:

```
namespace DTO;

class CreateUserDTO
{
    public string $email;
    public float $balance;
}
```

```
$data = [
    'email' => 'test@mail.com',
    'balance' => 128.41,
];
$dto = (new Hydrator())->create(CreateUserDTO::class, $data);

// or static init

$dto = Hydrator::init()->create(CreateUserDTO::class, $data);

var_dump($dto);
```

Output:

Also for php 8 you can pass named arguments:

```
$dto = Hydrator::init()->create(CreateUserDTO::class,
        email: 'test@mail.com',
        balance: 128.41
      );
```

If the property is not of a scalar type, but a class of another DTO is allowed, it will also be automatically converted.

```
class ProductDTO
{
    public int $id;
    public string $name;
}

class PurchaseDTO
{
```

(continues on next page)

(continued from previous page)

```
    public ProductDTO $product;
    public float $cost;
}

$data = [
    'product' => ['id' => 1, 'name' => 'phone'],
    'cost' => 10012.23,
];

$purchaseDTO = Hydrator::init()->create(PurchaseDTO::class, $data);
var_dump($purchaseDTO);
```

```
object(PurchaseDTO)
  public ProductDTO 'product' =>
    object(ProductDTO)
      public int 'id' => int 1
      public string 'name' => string 'phone' (length=5)
  public float 'cost' => float 10012.23
```

2.1 Collection

If you have an array of objects of a certain class, then you must specify the ConvertArray attribute for it, passing it to which class you need to bring the elements.

You can also specify a class in PHP DOC, but then you need to write the full path to this class *array <DTOProductDTO>*. This is done in order to know exactly which instance you need to create. Since Reflection does not provide out-of-the-box functions for getting the use * file. Besides use *, you can specify an alias, and it will be more difficult to trace it. Example:

2.2 Anonymous array

In case you need to convert an array of data into an array of class objects, you can implement this using the *createCollection* method.

```
$data = [
  ['id' => 1, 'name' => 'phone'],
  ['id' => 2, 'name' => 'bread'],
];
$products = Hydrator::init()->createCollection(ProductDTO::class, $data);
```

As a result of this execution, you will get an array of ProductDTO objects

```
array(2) {
  [0]=>
      object(ProductDTO) {
        ["id"]=> int(1)
        ["name"]=> string(5) "phone"
      }
  [1]=>
```

(continues on next page)

(continued from previous page)

```
      object(ProductDTO) {
        ["id"]=> int(2)
        ["name"]=> string(5) "bread"
      }
}
```

You may also need a piecemeal transformation of the array. In this case, you can pass an array of classes, which can then be easily unpacked.

```
$userData = ['id' => 1, 'email' => 'test@test.com', 'balance' => 10012.23];
$purchaseData = [
    'products' => [
        ['id' => 1, 'name' => 'phone',],
        ['id' => 2, 'name' => 'bread',],
    ],
    'user' => ['id' => 3, 'email' => 'fake@mail.com', 'balance' => 10012.23,],
];

$result = Hydrator::init()->createMultiple([UserDTO::class, PurchaseDTO::class], [
↪$userData, $purchaseData]);

[$user, $purchase] = $result;
var_dump($user);
var_dump($purchase);
```

Result:

```
object(UserDTO) (3) {
  ["id"] => int(1)
  ["email"]=> string(13) "test@test.com"
  ["balance"]=> float(10012.23)
}

object(PurchaseDTO) (2) {
  ["products"]=>
  array(2) {
    [0]=>
    object(ProductDTO)#349 (3) {
      ["id"]=> int(1)
      ["name"]=> string(5) "phone"
    }
    [1]=>
    object(ProductDTO)#348 (3) {
      ["id"]=> int(2)
      ["name"]=> string(5) "bread"
    }
  }
  ["user"]=>
  object(UserDTO)#332 (3) {
    ["id"]=> int(3)
    ["email"]=> string(13) "fake@mail.com"
    ["balance"]=> float(10012.23)
```

(continues on next page)

(continued from previous page)

```
  }
}
```

2.3 Writing style

A constant problem with the style of writing, for example, in the database it is snake_case, and in the camelCase code. And they constantly need to be transformed somehow. The package takes care of this, you just need to specify the WritingStyle attribute on the property:

```
class WritingStyleSnakeCaseDTO
{
    #[WritingStyle(WritingStyle::STYLE_CAMEL_CASE, WritingStyle::STYLE_SNAKE_CASE)]
    public string $contact_fio;

    #[WritingStyle(WritingStyle::STYLE_CAMEL_CASE)]
    public string $contact_email;
}


 $data = [
  'contactFio' => 'yzen.dev',
  'contactEmail' => 'test@mail.com',
];
$model = Hydrator::init()->create(WritingStyleSnakeCaseDTO::class, $data);
var_dump($model);
```

Output:

```
object(WritingStyleSnakeCaseDTO) (2) {
  ["contact_fio"]=> string(8) "yzen.dev"
  ["contact_email"]=> string(13) "test@mail.com"
}
```

2.4 Alias

Various possible aliases can be set for the property, which will also be searched in the data source. This can be useful if the DTO is generated from different data sources.

```
class WithAliasDTO
{
    #[FieldAlias('userFio')]
    public string $fio;

    #[FieldAlias(['email', 'phone'])]
    public string $contact;
}
```

2.5 Custom setter

If a field requires additional processing during its initialization, you can mutate its setter. To do this, create a method in the following format in the class - *set{$name}Attribute*. Example:

```
class UserDTO
{
    public int $id;
    public string $real_address;

    public function setRealAddressAttribute(string $value)
    {
        $this->real_address = strtolower($value);
    }
}
```

2.6 After Transform

Inside the class, you can create the *afterTransform* method, which will be called immediately after the conversion is completed. In it, we can describe our additional verification or transformation logic by already working with the state of the object.

```
class UserDTO
{
    public int $id;
    public float $balance;

    public function afterTransform()
    {
        $this->balance = 777;
    }
}
```

2.7 Custom transform

If you need to completely transform yourself, then you can create a transform method in the class. In this case, no library processing is called, all the responsibility of the conversion passes to your class.

```
class CustomTransformUserDTOArray
{
    public string $email;
    public string $username;

    public function transform($args)
    {
        $this->email = $args['login'];
        $this->username = $args['fio'];
    }
}
```

2.8 Cache

The package supports a class caching mechanism to avoid the cost of reflection. This functionality is recommended to be used only if you have very voluminous classes, or there is a cyclic transformation of multiple entities. On ordinary lightweight DTO, there will be only 5-10%, and this will be unnecessary access in the file system.

You can enable caching by passing the config to the hydrator constructor:

```
(new Hydrator(new HydratorConfig(true)))
    ->create(PurchaseDto::class, $data);
```