Como criar um container de injecção de dependência em PHP

Container é uma classe que gerência a injecção de dependência e a resolução de dependência.

1. Primeiro é necessário garantir que a instância container seja única.

```php
public static  function getInstance() : Container
   {
       static $instance = null;
       if(is_null($instance)){$instance = new self();}
       return $instance;
   }
```

2. Em seguida, é necessário criar um método para registrar instâncias que não são instanciadas automaticamente.

```php
private array $registry = [];

public function registerInstance($className, Closure $value) : array{
      $this->registry[$className] = $value;
      return $this->registry;
  }
```

3. Criar um método para resolver as dependências automaticamente de forma recursiva isto é, se a classe A depende da classe B e a classe B depende da classe C, o container deve conseguir resolver a classe C e a classe B antes de resolver a classe A.

```php
...
$resolvedClassA =  new ClassA(new ClassB(new ClassC()));
...
```

3.1 O método responsável por resolver as dependências de uma classe.

```php
   public function get(string $classname){

       # se a classe já foi registrada, então retorna a instância sem resolver as dependência
       if(isset($this->registry[$classname])){

           return $this->registry[$classname]();
```

```
        }

        # se a classe não foi registrada, então resolve as dependências

        $reflection = new ReflectionClass($classname);

        $constructor = $reflection->getConstructor();

        if(is_null($constructor)){
            return new $classname;
        }

        # Se a classe tem um construtor, então resolve as dependências do construtor recursiva
        # e instancia a classe com as dependências resolvidas
        $deps = [];
        foreach ($constructor->getParameters() as $parameter) {
            $parameterType = $parameter->getType();
            $deps[] = $this->get($parameterType);
          }
        return $reflection->newInstanceArgs($deps);
    }
```

5. A classe final ficará assim:

```
<?php

namespace Mutane\DIContainer;

use Closure;
use ReflectionClass;
use ReflectionException;

class Container
{
    private array $registry = [];

    public static  function getInstance() : Container
    {
        static $instance = null;
        if(is_null($instance)){$instance = new self();}
        return $instance;
    }

    public function registerInstance($className, Closure $value) : array{
        $this->registry[$className] = $value;
        return $this->registry;
    }

    /**
     * @throws ReflectionException
     */
    public function get(string $classname){

        if(isset($this->registry[$classname])){
```

```
            return $this->registry[$classname]();

        }

        $reflection = new ReflectionClass($classname);

        $constructor = $reflection->getConstructor();

        if(is_null($constructor)){
            return new $classname;
        }

        $deps = [];
        foreach ($constructor->getParameters() as $parameter) {
            $parameterType = $parameter->getType();
            $deps[] = $this->get($parameterType);
          }
        return $reflection->newInstanceArgs($deps);
    }
}
```

Exemplo de uso do container

Suponha que temos as seguintes classes:

UserDto.php

```
<?php

namespace Mutane\Dto;

readonly class UserDto
{
    public function __construct(
        public int    $id,
        public string $name,
        public string $email,
    ){}
}
```

Database.php

```
<?php

namespace Mutane\Contracts;

use Mutane\Dto\UserDto;
```

```php
interface Database
{
    /**
     * @return UserDto[]
     */
    public function fetchUsers() : array;
}
```

UserRepository.php

```php
<?php

namespace Mutane\Repository;

use Mutane\Contracts\Database;
use Mutane\Dto\UserDto;

readonly class UserRepository
{
    public function __construct(private Database $database){}

    /**
     * @return UserDto[]
     */
    public function getUsers() : array
    {
        $users = $this->database->fetchUsers();
        $transformedUsers = [];
        foreach ($users as $user) {
            $transformedUsers[] = new UserDto(
                $user['id'],
                $user['name'],
                $user['email']
            );
        }
        return $transformedUsers;
    }
}
```

Suponha que temos a seguinte implementação da interface Database:

ArrayDatabase.php

```php
<?php

namespace Mutane\Database;

use Mutane\Contracts\Database;

class ArrayDatabase implements Database
{
```

```
    /**
     * @inheritDoc
     */
    public function fetchUsers(): array
    {
       return [
           ['id' => 1, 'name' => 'John Doe', 'email' => 'johndoe@array.mutane.me'],
           ['id' => 2, 'name' => 'Jane Doe', 'email' => 'janedoe@array.mutane.me']
       ];
    }
}
```

JsonDatabase.php

```
<?php

namespace Mutane\Database;
use Exception;
use Mutane\Contracts\Database;

class JsonDatabase implements Database
{
    public function fetchUsers(): array
    {
        $data =  file_get_contents(__DIR__ .'/users.json');
        try {
         return    json_decode($data, true, 512, JSON_THROW_ON_ERROR);
        }catch (Exception $e){
            var_dump($e->getMessage());
            die;
        }
    }
}
```

Sendo esse o users.json

```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "johnDoe@json.mutane.me"
  },
    {
        "id": 2,
        "name": "Jane Doe",
        "email": "JaneDoe@json.mutane.me"
    }
]
```

Então, podemos usar o container para resolver as dependências automaticamente da seguinte forma:

```php
$container = Container::getInstance();

$container->registerInstance(
   className : Database::class,
   value     : fn() => new ArrayDatabase #JsonDatabase
);

$repository = $container->get(
    className: UserRepository::class
);
```