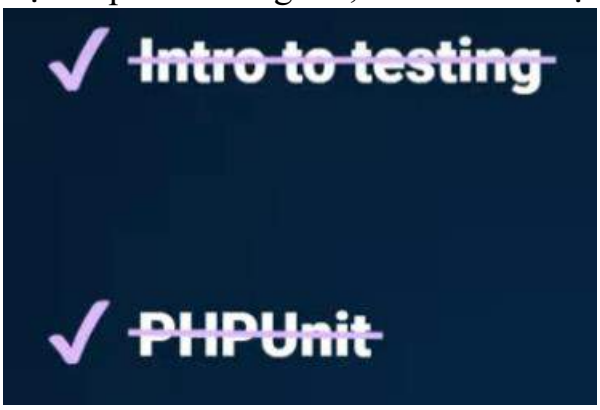
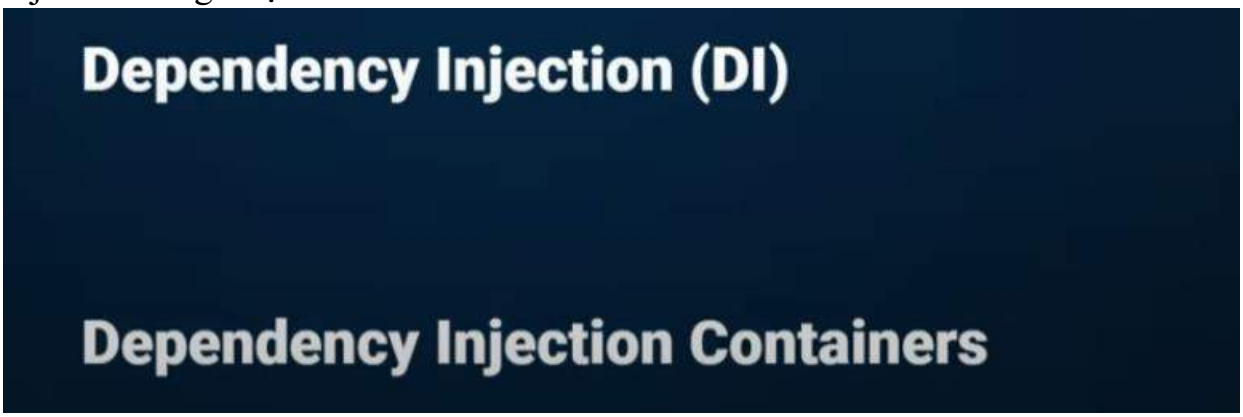
3.03 - DEPENDENCY INJECTION & DI CONTAINERS

- Chúng ta đã tìm hiểu về cơ bản của việc kiểm thử với PHPUnit và viết một số bài kiểm tra đơn vị.
- Sử dụng "dependency injection" thay vì sử dụng các phụ thuộc được cố định trực tiếp vào mã nguồn, để làm cho việc kiểm thử dễ dàng hơn.



- Video đi sâu hơn vào việc "dependency injection" cùng với "dependency injection containers", "auto wiring", và "reflection API". Vậy "dependency injection" là gì một cách chính thức.



In software engineering, **dependency injection** is a technique in which an object receives other objects that it depends on, called dependencies. Typically, the receiving object is called a client and the passed-in ('injected') object is called a service. The code that passes the service to the client is called the injector. Instead of the client specifying which service it will use, the injector tells the client what service to use. The 'injection' refers to the passing of a dependency (a service) into the client that uses it.

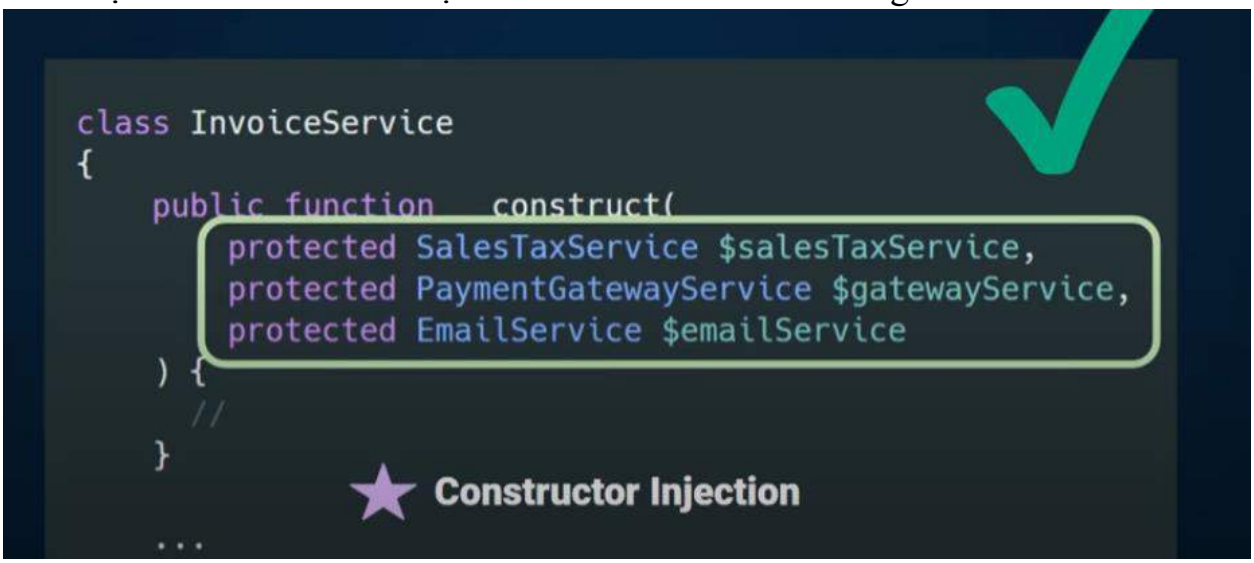
- Trong kỹ thuật kỹ thuật phần mềm, "dependency injection" là một kỹ thuật trong đó một đối tượng nhận các đối tượng khác mà nó phụ thuộc vào, được gọi là các phụ thuộc. Thông thường, đối tượng nhận được này được gọi là một khách hàng và đối tượng được chuyển vào (được "tiêm vào") được gọi là một dịch vụ. Mã nguồn thực hiện việc chuyển dịch vụ vào khách hàng được gọi là "injector". Thay vì khách hàng chỉ định dịch vụ nào nó sẽ sử dụng, injector cho biết cho khách hàng nên sử dụng dịch vụ nào. Thuật ngữ "injection" (tiêm) liên quan đến việc truyền một phụ thuộc (một dịch vụ) vào khách hàng mà sử dụng nó.

- Cố định (hard code) các lớp (classes) trong hàm tạo (constructor), trong đó lớp InvoiceService (dịch vụ hóa đơn) chịu trách nhiệm tạo ra các phụ thuộc (dependencies).

```
class InvoiceService
{
    protected PaymentGatewayService $gatewayService;
    protected SalesTaxService       $salesTaxService;
    protected EmailService          $emailService;

    public function __construct()
    {
        $this->gatewayService  = new PaymentGatewayService();
        $this->salesTaxService = new SalesTaxService();
        $this->emailService    = new EmailService();
    }

    ...
```

- Điều này tạo ra sự ràng buộc chặt chẽ (tight coupling), làm cho mã nguồn khó bảo trì hơn và khó kiểm thử hơn. Thay vì lớp InvoiceService (lớp dịch vụ hóa đơn) chịu trách nhiệm tạo ra các phụ thuộc của nó

- Có thể đảo ngược luồng điều khiển và truyền các phụ thuộc thông qua "constructor dependency injection". Theo cách cơ bản, đây là một hình thức của "inversion of control" (đảo ngược điều khiển), hoặc viết tắt là IOC. IOC là một nguyên tắc tổng quan và có thể được áp dụng cho nhiều thứ, không chỉ riêng việc tiêm phụ thuộc. Với IOC, luồng điều khiển được đảo ngược cơ bản, thay vì lớp tự quyết định các phụ thuộc của nó và tạo ra các đối tượng.

- Phần này gọi là "constructor injection. "Dependency injection" cùng với nhiều lợi ích khác làm cho việc viết các bài kiểm tra dễ dàng hơn
- 

- có thể tạo ra các "doubles" (giả lập) và truyền chúng như là các phụ thuộc thay vì sử dụng các lớp gốc, điều này đã làm cho việc kiểm thử dễ dàng hơn

nhiều. có thể sử dụng các giao diện thay vì chấp nhận các lớp cụ thể làm phụ thuộc.

```php
/** @test */
public function it_processes_invoice(): void
{
    $salesTaxServiceMock = $this->createMock(SalesTaxService::class);
    $gatewayServiceMock  = $this->createMock(PaymentGatewayService::class);
    $emailServiceMock    = $this->createMock(EmailService::class);

    $invoiceService = new InvoiceService(
        $salesTaxServiceMock,
        $gatewayServiceMock,
        $emailServiceMock
    );

    ...
}
```

- Việc thực hiện một giao diện cho phép chúng ta thay thế các phiên bản thực thi một cách dễ dàng theo nhu cầu trong thời gian chạy (runtime).

Vai trò của Depency injection

- Các "dependency injection containers" đóng một vai trò quan trọng. Hãy thu nhỏ phạm vi ra khỏi lớp InvoiceService và di chuyển lên một cấp độ, nơi chúng ta cần lớp InvoiceService chính nó. Điều này có thể là một bộ điều khiển (controller) hoặc bất kỳ lớp nào khác cần một đối tượng của lớp InvoiceService.

- Hãy giả sử chúng ta có một bộ điều khiển (controller) Invoice với một phương thức store để xử lý hóa đơn sau khi người dùng gửi một biểu mẫu. Theo cách đơn giản nhất, người dùng điền thông tin vào một biểu mẫu, và chúng tôi chỉ sử dụng trường tên và số lượng (name và amount) trong ví dụ này. Tất nhiên, trong một ứng dụng thực tế, có thể có nhiều trường hơn như số thẻ tín dụng, địa chỉ thanh toán và nhiều thông tin khác, nhưng điều này không phải là điểm quan trọng. Phương thức store là nơi chúng ta cần đối tượng của lớp InvoiceService vì chúng ta cần xử lý hóa đơn. Tuy nhiên, chúng ta đang tạo một đối tượng mới của lớp InvoiceService ở đây, nhưng vì chúng ta phải truyền các phụ thuộc một cách thủ công.

```php
class InvoiceController
{
    public function store()
    {
        $name   = $_GET['name'];
        $amount = $_GET['amount'];

        $invoiceService = new InvoiceService(
            new SalesTaxService(),
            new PaymentGatewayService(),
            new EmailService()
        );

        $invoiceService->process(['name' => $name], $amount);
    }
}
```

- Chúng ta cũng đang tạo ra các phụ thuộc này và truyền chúng xuống, bạn có thể thấy vấn đề ở đây không? Trong khi lớp InvoiceService được kết nối lỏng lẻo và sử dụng "dependency injection", thì bộ điều khiển (controller) lại kết nối chặt chẽ và chúng ta vẫn đang cố định các phụ thuộc. Nhưng bây giờ, ngoài việc cố định ba phụ thuộc này,

```
    new SalesTaxService(),
    new PaymentGatewayService(),
    new EmailService()
```

- Chúng ta cũng đang cố định lớp InvoiceService chính nó như một phụ thuộc.

```
$invoiceService = new InvoiceService(
    new SalesTaxService(),
    new PaymentGatewayService(),
    new EmailService()
);
```

```php
class InvoiceController
{
    public function store()
    {
        $name   = $_GET['name'];
        $amount = $_GET['amount'];

        $invoiceService = new InvoiceService(
            new SalesTaxService(),
            new PaymentGatewayService(
                new ApiService(),
                new Logger()
            ),
            new EmailService()
        );

        $invoiceService->process(['name' => $name], $amount);
    }
}
```

- Để khắc phục vấn đề này và sử dụng "dependency injection" trong bộ điều khiển (controller) cũng như trong các lớp phụ thuộc, chúng ta cần thực hiện các thay đổi.

```php
    new SalesTaxService(),
    new PaymentGatewayService(
        new ApiService(),
        new Logger()
    ),
    new EmailService()
);
```

- Bây giờ, chúng ta đang chấp nhận lớp InvoiceService trong hàm tạo và phương thức store của chúng ta trông gọn gàng hơn nhiều. Tuy nhiên

```php
class InvoiceController
{
    public function __construct(
      private InvoiceService $invoiceService
    ) {

    }

    public function store()
    {
        $name   = $_GET['name'];
        $amount = $_GET['amount'];

        $this->invoiceService->process(
          ['name' => $name],
          $amount
        );
    }
}
```

- đã tạo một lớp bộ định tuyến (router) cơ bản, nhiệm vụ của nó là giải quyết phần về bộ điều khiển, ánh xạ các tuyến đường (routes) với các hành động của bộ điều khiển.

```php
public function resolve(string $requestUri, string $requestMethod)
{
    $route = explode('?', $requestUri)[0];
    $action = $this->routes[$requestMethod][$route] ?? null;

    if (! $action) {
        throw new RouteNotFoundException();
    }

    if (is_callable($action)) {
        return call_user_func($action);
    }

    [$class, $method] = $action;

    if (class_exists($class)) {
        $class = new $class();

        if (method_exists($class, $method)) {
            return call_user_func_array([$class, $method], []);
        }
    }

    throw new RouteNotFoundException();
}
```

- Đúng, ở đây, kỹ thuật, đối tượng của bộ điều khiển được tạo ra, và chúng ta thực sự không thể truyền các phụ thuộc được cố định ở đây, vì các bộ điều khiển khác có thể có các phụ thuộc khác nhau hoặc thậm chí không có phụ thuộc nào cả. Nhưng hãy quên về bộ điều khiển hoặc bộ định tuyến một chút. Ngay cả khi chúng ta có cách truyền các phụ thuộc xuống, vấn đề vẫn giống như trước đó, chúng ta sẽ phải tạo thủ công tất cả các phụ thuộc này và truyền chúng xuống thủ công ở một nơi cao hơn trong cây phụ thuộc. Bây giờ có thể bạn đang hét lên: "Giải pháp là gì? Hãy điểm qua đi!"

```php
$class = new $class();
```

- the point is this is where dependency injection containers come into play dependency injection container simply put is just a class that has information about other classes which allows us to replace

- Vấn đề chính ở đây là "dependency injection containers" đóng vai trò quan trọng. Một "dependency injection container" đơn giản là một lớp chứa thông tin về các lớp khác, cho phép chúng ta thay thế các phụ thuộc một cách dễ dàng.

- Điều quan trọng là, khi chúng ta cần một đối tượng của lớp InvoiceService, chúng ta chỉ cần yêu cầu "container" cung cấp cho chúng ta đối tượng đó và để cho "container" xác định cách giải quyết các phụ thuộc một cách đúng đắn. Có nhiều phương pháp và cách thức khác nhau để thực hiện các "dependency injection containers".

- Có một tiêu chuẩn gọi là PSR-11, được thiết kế để cung cấp hướng dẫn hoặc một giao diện chung có thể được thực hiện vào các framework, thư viện và mã nguồn.

3.04) Dependency Injection Container With & Without Reflection API Autowiring

- Thực sự thực hiện một "dependency injection container" đơn giản. Nếu bạn chưa xem bài học trước hoặc cần làm mới kiến thức về "dependency injection" và "dependency injection containers," hãy xem bài học đó trước, liên kết sẽ có trong phần mô tả hoặc bạn có thể tìm thấy nó trong danh sách phát.

- Về cơ bản, "dependency injection container" (thùng chứa phụ thuộc) đơn giản là một lớp có thông tin về các lớp khác, cho phép nó giải quyết các lớp với các phụ thuộc của chúng. Nếu chúng ta nhìn vào biểu đồ từ bài học trước, với một "dependency injection container" đúng đắn, chúng ta có thể thay thế đoạn mã này

```
$invoiceService = new InvoiceService(
    new SalesTaxService(),
    new PaymentGatewayService(
        new ApiService(),
        new Logger()
    ),
    new EmailService()
);
```

- bằng đoạn mã này.

```
$invoiceService = $container->get(InvoiceService::class);
```

- Nhưng để làm việc này, cần xây dựng "dependency injection container" và cho nó biết cách giải quyết các lớp và phụ thuộc của chúng.

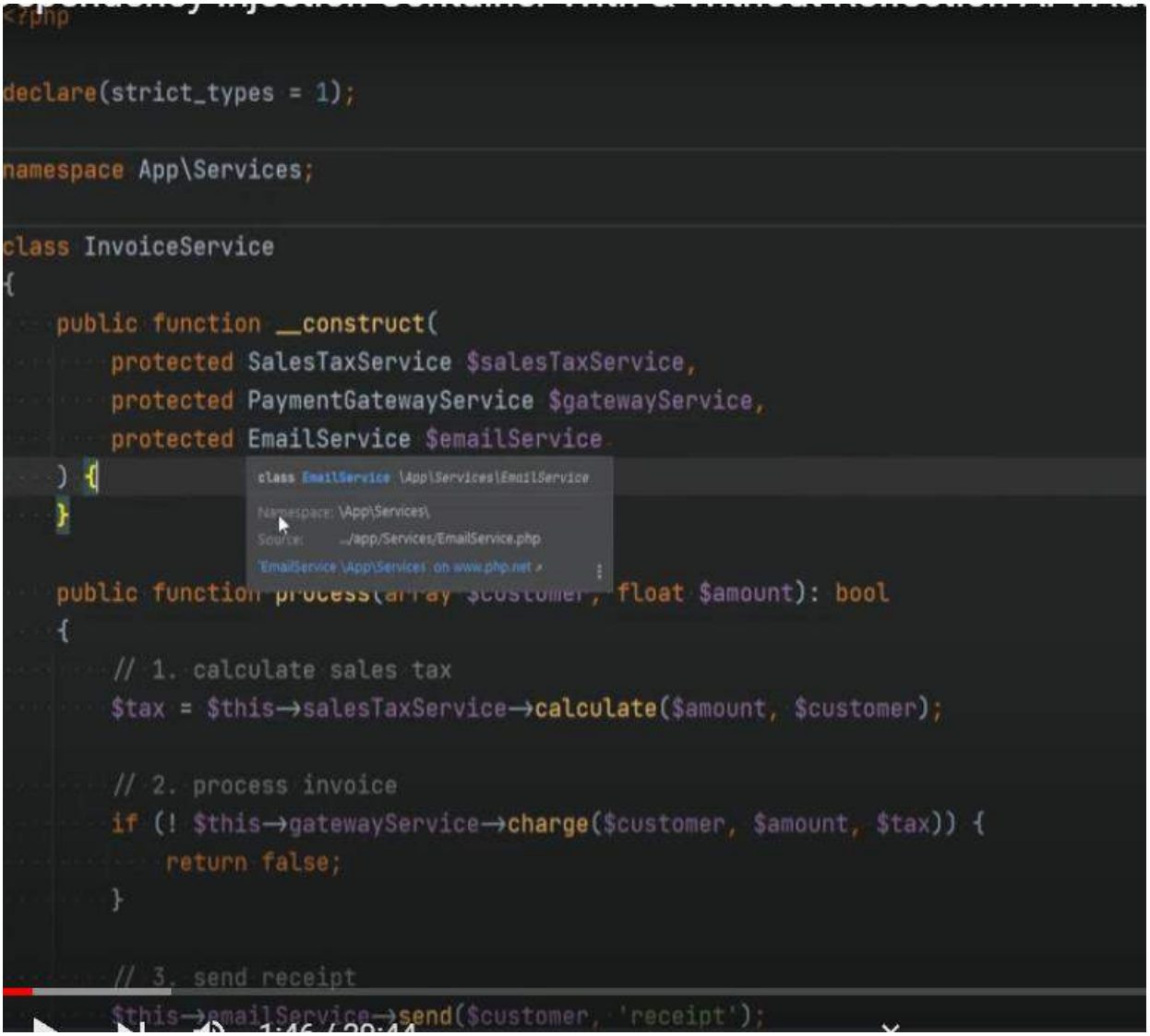
- sử dụng PSR-11 để thực hiện một "dependency injection container" đơn giản, PSR-11 cung cấp các giao diện và hướng dẫn cần thiết. PSR-11 là tiêu chuẩn mà hầu hết các framework và thư viện sử dụng để cung cấp chức năng "dependency injection container" với một số tính năng bổ sung. Hiện đã có

rất nhiều "dependency injection containers" được triển khai dựa trên PSR-11.

- Có rất nhiều triển khai (implementations) của "dependency injection containers" mà bạn có thể sử dụng, như PHP-DI,

- hoặc nếu sử dụng một framework, có khả năng rất cao rằng framework đó đã cung cấp một triển khai cho "dependency injection container" ngay từ đầu, ví dụ như Laravel.

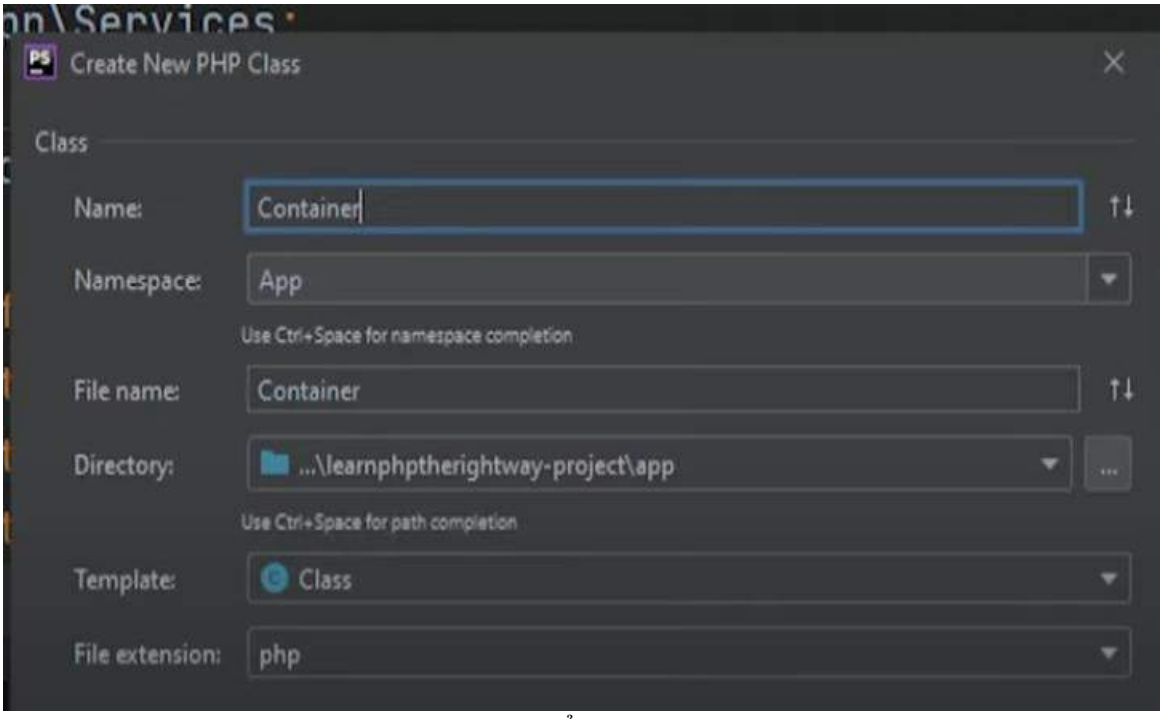
- Vậy thì, chúng ta hãy chuyển sang mã nguồn. Từ bài học trước, chúng ta có lớp InvoiceService với ba phụ thuộc trong hàm tạo: SalesTaxService, PaymentGatewayService, và EmailService. Chúng ta sẽ thực hiện một "dependency injection container" đơn giản để giúp giải quyết các phụ thuộc này. Đầu tiên, chúng ta cần cài đặt gói phụ thuộc "psr11/container" thông qua Composer.



- Mở cửa sổ terminal và gõ lệnh sau để cài đặt gói Composer "psr/container":

```
root@6755bac6e880:/var/www# composer require psr/container
```

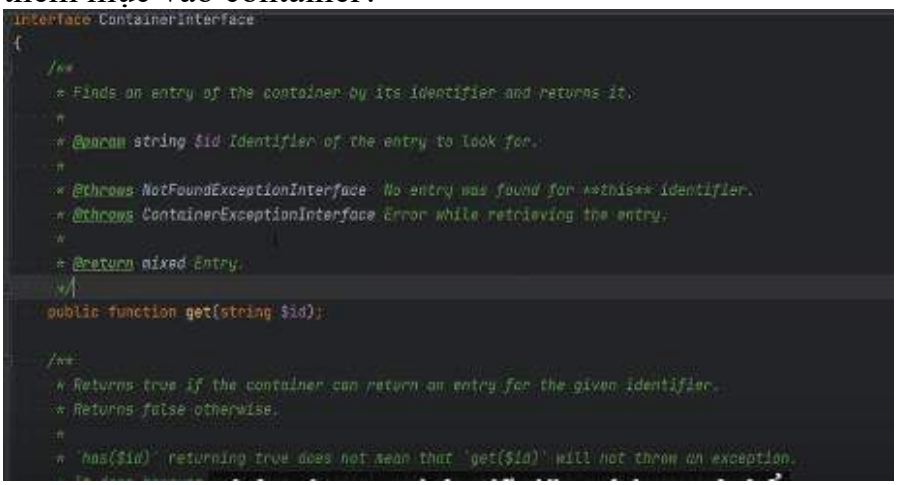
- Sau khi đã cài đặt xong, sẽ tạo một lớp mới có tên là "Container."



- Chúng ta sẽ cài đặt lớp "Container" để tuân thủ giao diện "Container" từ gói PSR-11. Lớp này cần triển khai hai phương thức là "get" và "has." Hãy thêm các phương thức này vào lớp Container.

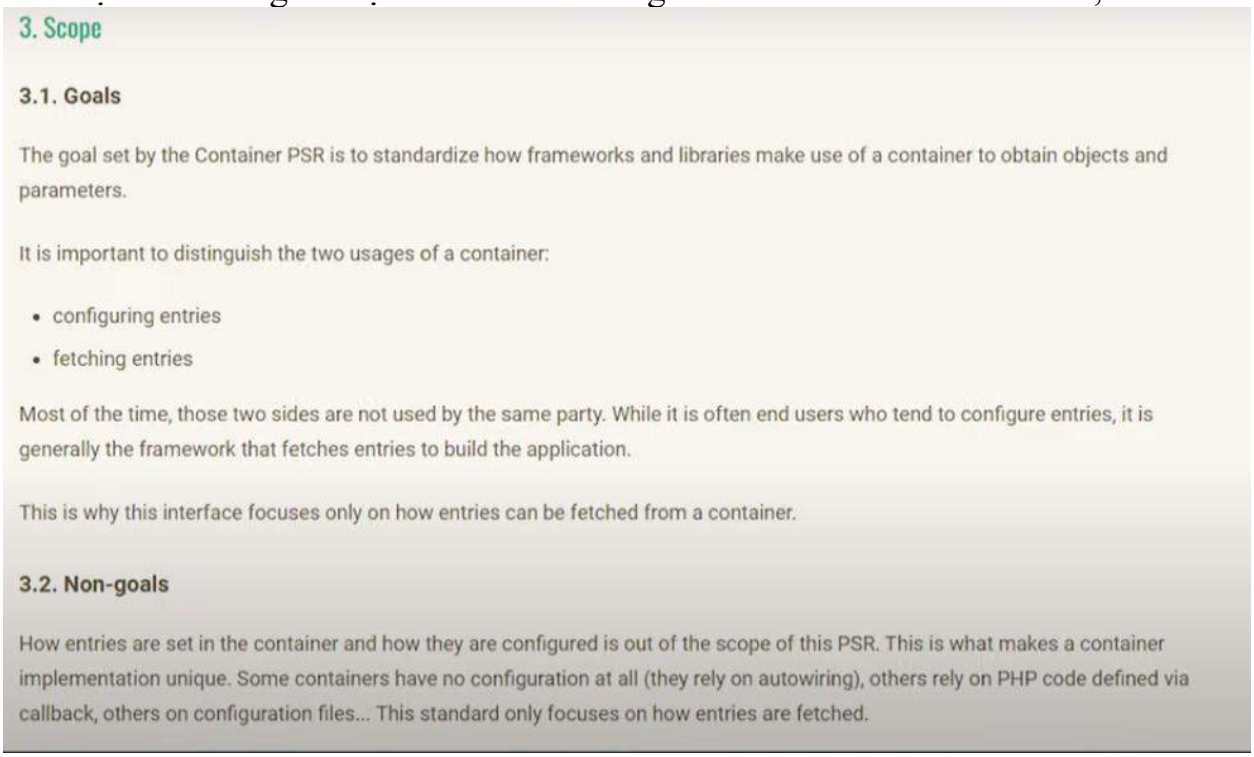
```php
declare(strict_types = 1);

namespace App;

use Psr\Container\ContainerExceptionInterface;
use Psr\Container\ContainerInterface;
use Psr\Container\NotFoundExceptionInterface;

class Container implements ContainerInterface
{
    public function get(string $id)
    {
        // TODO: Implement get() method.
    }

    public function has(string $id): bool
    {
        // TODO: Implement has() method.
    }
}
```

- Hãy nhấp vào giao diện "Container" để xem các phương thức này là gì. Phương thức "get" được sử dụng để lấy lớp hoặc giải quyết lớp từ container, và phương thức "has" được sử dụng để kiểm tra xem có một liên kết hoặc một lớp hoặc mục nào đó tồn tại trong container. Đối số "id" ở đây đề cập tên lớp đầy đủ (fully qualified class name) mà bạn đang cố gắng giải quyết.
- Nhưng đợi một chút, bạn có thể đặt câu hỏi rằng nơi đây không có phương thức "set." Giao diện chỉ định các phương thức "get" và "has," nhưng không có phương thức "set." Làm thế nào chúng ta có thể đặt các liên kết hoặc thêm mục vào container?



```
interface ContainerInterface
{
    /**
     * Finds an entry of the container by its identifier and returns it.
     *
     * @param string $id Identifier of the entry to look for.
     *
     * @throws NotFoundExceptionInterface  No entry was found for **this** identifier.
     * @throws ContainerExceptionInterface Error while retrieving the entry.
     *
     * @return mixed Entry.
     */
    public function get(string $id);

    /**
     * Returns true if the container can return an entry for the given identifier.
     * Returns false otherwise.
     *
     * `has($id)` returning true does not mean that `get($id)` will not throw an exception.
```

- Nếu bạn xem trang tài liệu "meta" trên trang web PSR-11 documentation,



3. Scope

3.1. Goals

The goal set by the Container PSR is to standardize how frameworks and libraries make use of a container to obtain objects and parameters.

It is important to distinguish the two usages of a container:

- configuring entries
- fetching entries

Most of the time, those two sides are not used by the same party. While it is often end users who tend to configure entries, it is generally the framework that fetches entries to build the application.

This is why this interface focuses only on how entries can be fetched from a container.

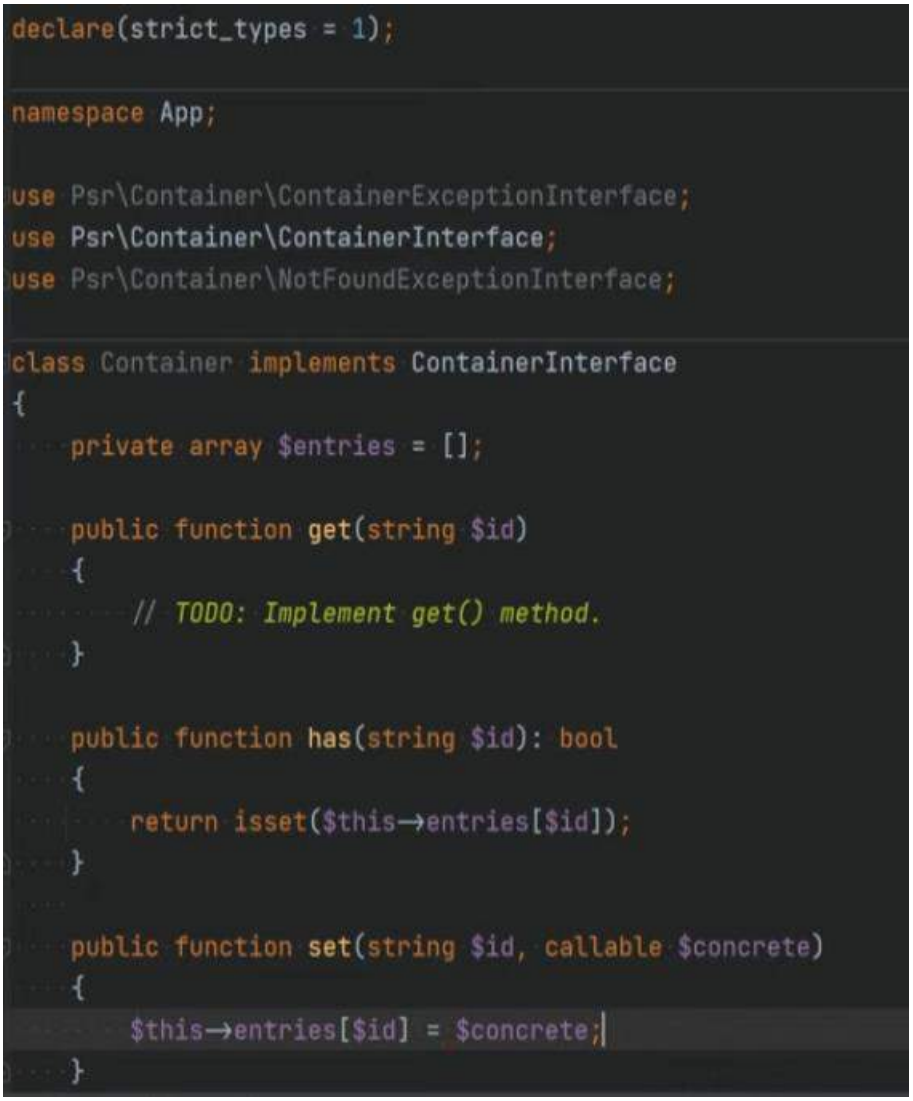
3.2. Non-goals

How entries are set in the container and how they are configured is out of the scope of this PSR. This is what makes a container implementation unique. Some containers have no configuration at all (they rely on autowiring), others rely on PHP code defined via callback, others on configuration files... This standard only focuses on how entries are fetched.

- Điều quan trọng là PSR-11 chỉ định hai phương thức cơ bản, "get" để giải quyết một mục và "has" để kiểm tra sự tồn tại của một mục trong container. Cách bạn thực hiện lưu trữ mục cụ thể có thể khác nhau tùy thuộc vào framework hoặc thư viện mà bạn đang sử dụng.

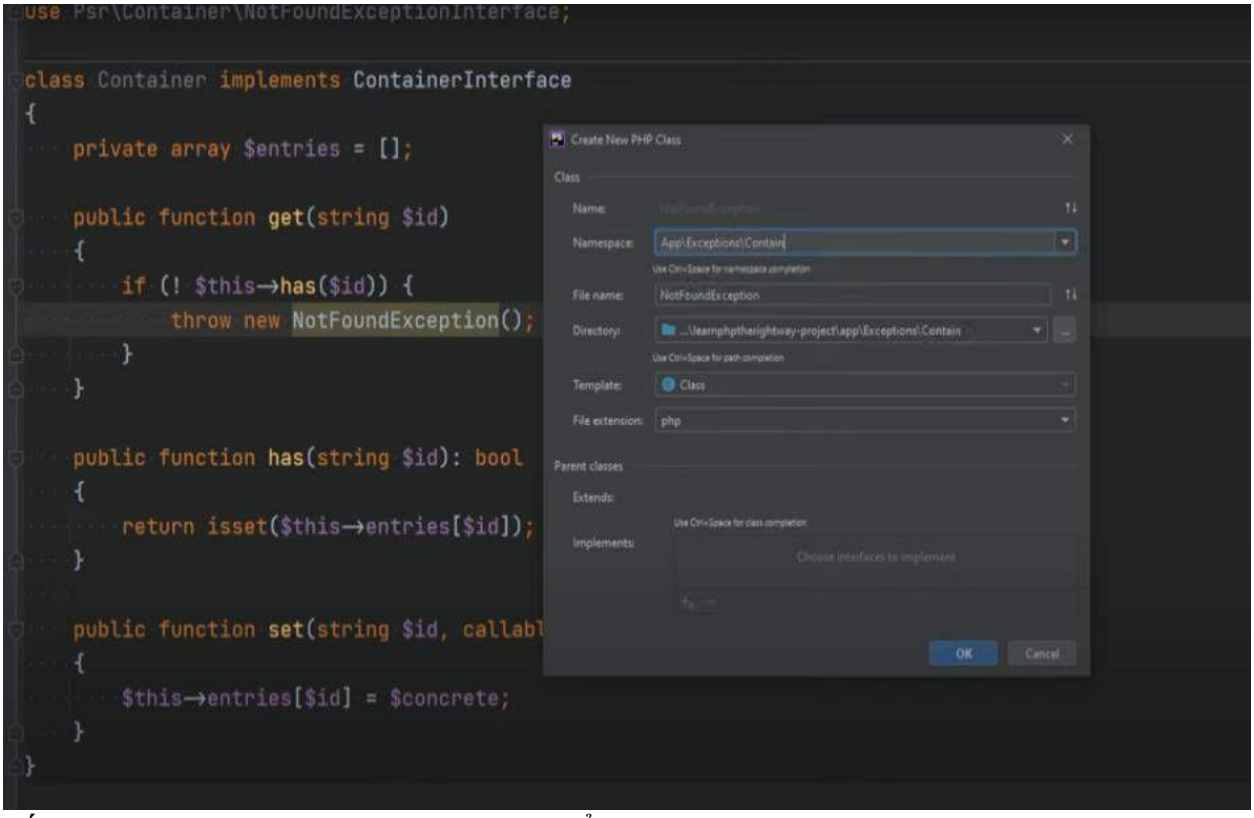
> How entries are set in the container and how they are configured is out of the scope of this PSR. This is what makes a container implementation unique. Some containers have no configuration at all (they rely on autowiring), others rely on PHP code defined via callback, others on configuration files... This standard only focuses on how entries are fetched.

- Một "dependency injection container" đơn giản là một lớp có thông tin về các lớp khác, biết rằng nó cần ít nhất hai phương thức: "get" và "has." Cấu trúc dữ liệu mà có thể sử dụng để lưu trữ các mục hoặc liên kết có thể là một mảng. Vì vậy, chúng ta sẽ định nghĩa một thuộc tính riêng tư là "entries" và đặt nó thành một mảng rỗng theo mặc định.
- Đầu tiên, có thể triển khai phương thức "has" bởi vì chúng ta biết rằng phương thức này sẽ trả về true nếu mục tồn tại và false nếu không tồn tại, có thể sử dụng hàm isset để làm điều này.
- Tiếp theo, chúng ta sẽ triển khai phương thức "set." Phương thức này sẽ chấp nhận hai đối số. Đối số đầu tiên là "id" (định danh), đó là tên lớp đầy đủ (fully qualified class name) mà chúng ta muốn giải quyết. Đối số thứ hai là triển khai hoặc giải quyết của lớp đó. Ở đây, chúng ta sử dụng liên kết dựa trên callback và định kiểu nó là callable. Chúng ta cũng có thể nghĩ về đối số thứ hai như một hàm "factory" chịu trách nhiệm tạo đối tượng của lớp nào đó mà bạn muốn lấy từ container.
- Triển khai của phương thức "set" khá đơn giản đối với "dependency injection container" đơn giản vì chỉ cần thêm mục này vào mảng "entries."

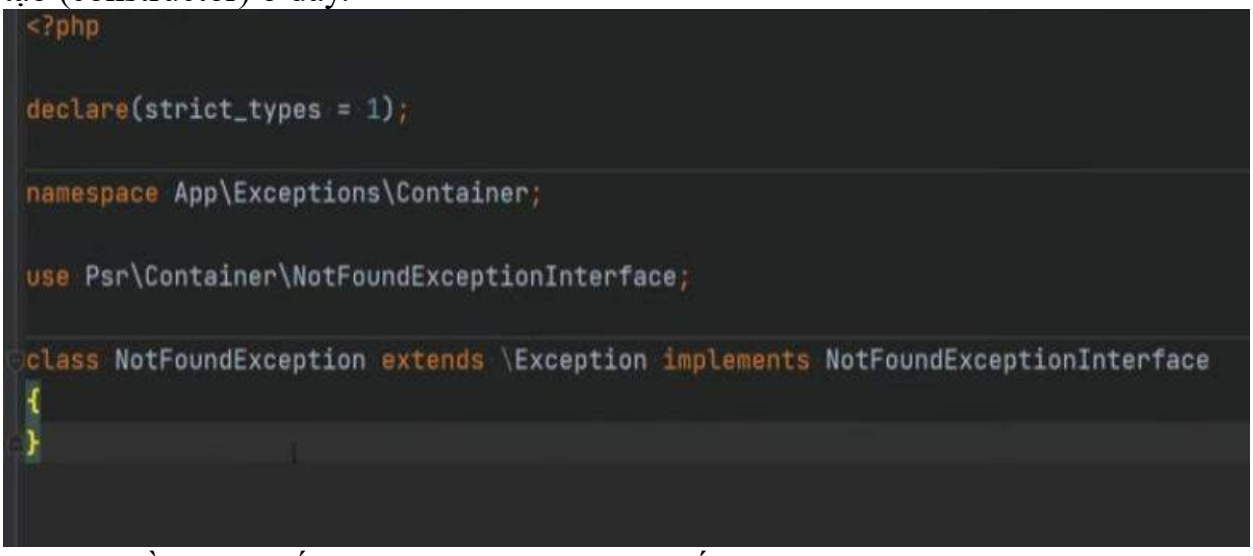


```php
declare(strict_types = 1);

namespace App;

use Psr\Container\ContainerExceptionInterface;
use Psr\Container\ContainerInterface;
use Psr\Container\NotFoundExceptionInterface;

class Container implements ContainerInterface
{
    private array $entries = [];

    public function get(string $id)
    {
        // TODO: Implement get() method.
    }

    public function has(string $id): bool
    {
        return isset($this->entries[$id]);
    }

    public function set(string $id, callable $concrete)
    {
        $this->entries[$id] = $concrete;
    }
```

- Cuối cùng, chúng ta cần triển khai phương thức "get" theo đúng quy định của PSR-11. Theo quy định này, nếu mục không được tìm thấy, phải ném một ngoại lệ "not found exception." Chúng ta có thể sử dụng phương thức "has" ở đây và nếu nó trả về false, chúng ta sẽ ném ngoại lệ.
- PSR-11 cũng cung cấp các giao diện cho các ngoại lệ, vì vậy chúng ta có thể tạo ra các ngoại lệ tùy chỉnh của riêng mình và triển khai các giao diện đó. Chúng ta có thể ném một ngoại lệ tên là "NotFoundException" (ngoại lệ không tìm thấy) ở đây.
- Để làm điều này, chúng ta cần tạo một lớp có tên "NotFoundException" trong một không gian tên (namespace) "app\exceptions\container." Bạn cần tạo thư mục "container" trong "app\exceptions" và thêm lớp "NotFoundException" vào đó.



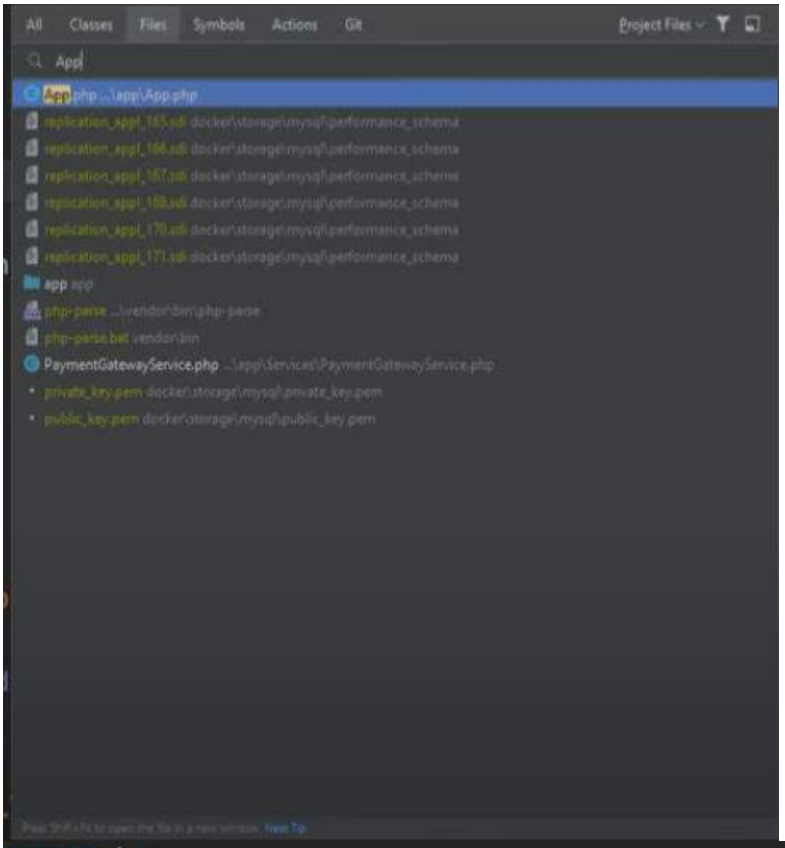
- kế thừa từ lớp cơ sở "Exception" và triển khai giao diện "NotFoundExceptionInterface" từ PSR Container. Không cần triển khai hàm tạo (constructor) ở đây.



- Nhưng điều còn thiếu là việc thêm hoặc liên kết các mục hoặc lớp vào container. Chúng ta sẽ đăng ký các liên kết hoặc mục nhập này trong lớp "app" để ví dụ này.

```php
use Psr\Container\ContainerInterface;
use Psr\Container\NotFoundExceptionInterface;

class Container implements ContainerInterface
{
    private array $entries = [];

    public function get(string $id)
    {
        if (! $this->has($id)) {
            throw new NotFoundException('Class "' . $id . '" has no binding');
        }

        $entry = $this->entries[$id];

        return $entry($this);
    }

    public function has(string $id): bool
    {
        return isset($this->entries[$id]);
    }

    public function set(string $id, callable $concrete): void
    {
        $this->entries[$id] = $concrete;
    }
```

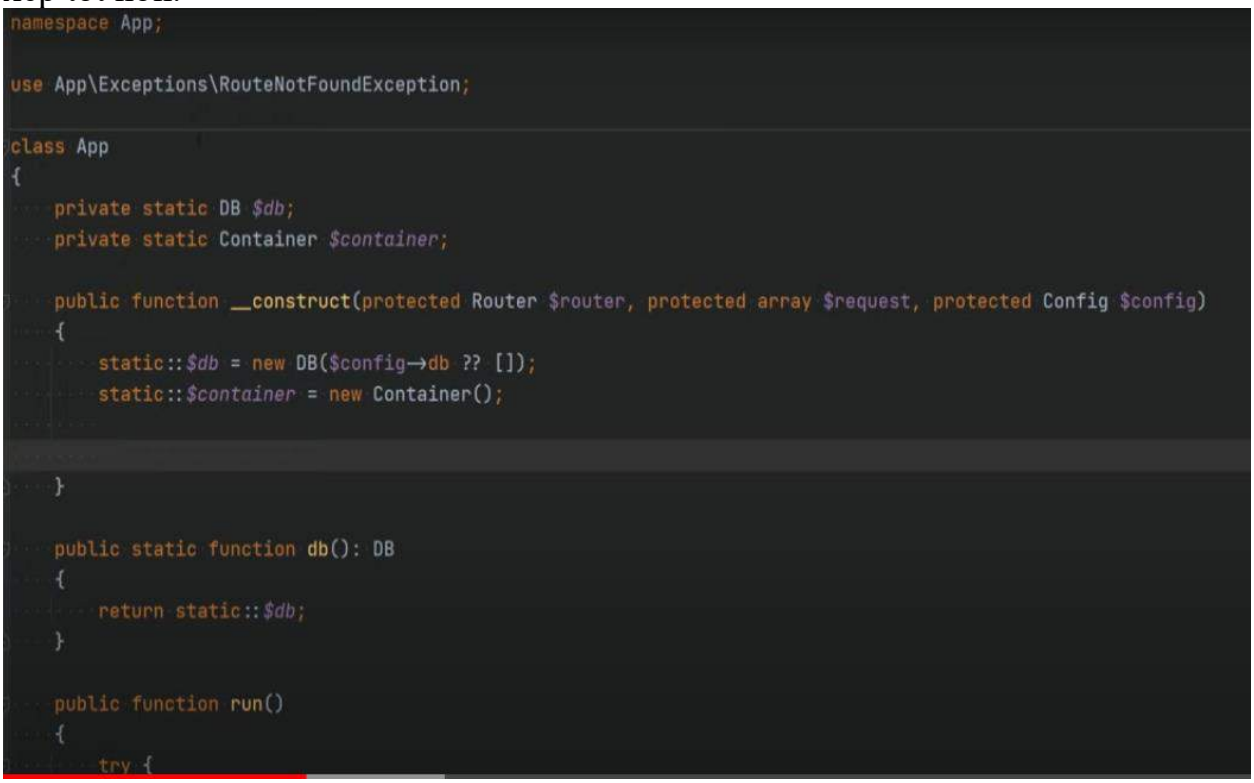
- Vậy thì, chúng ta sẽ mở lớp "app" để tiếp tục.




```php
namespace App;

use App\Exceptions\RouteNotFoundException;

class App
{
    private static DB $db;

    public function __construct(protected Router $router, protected array $request, protected Config $config)
    {
        static::$db = new DB($config->db ?? []);
    }

    public static function db(): DB
    {
        return static::$db;
    }

    public function run()
    {
        try {
            echo $this->router->resolve($this->request['uri'], strtolower($this->request['method']));
        } catch (RouteNotFoundException) {
            http_response_code(404);

            echo View::make('error/404');
```

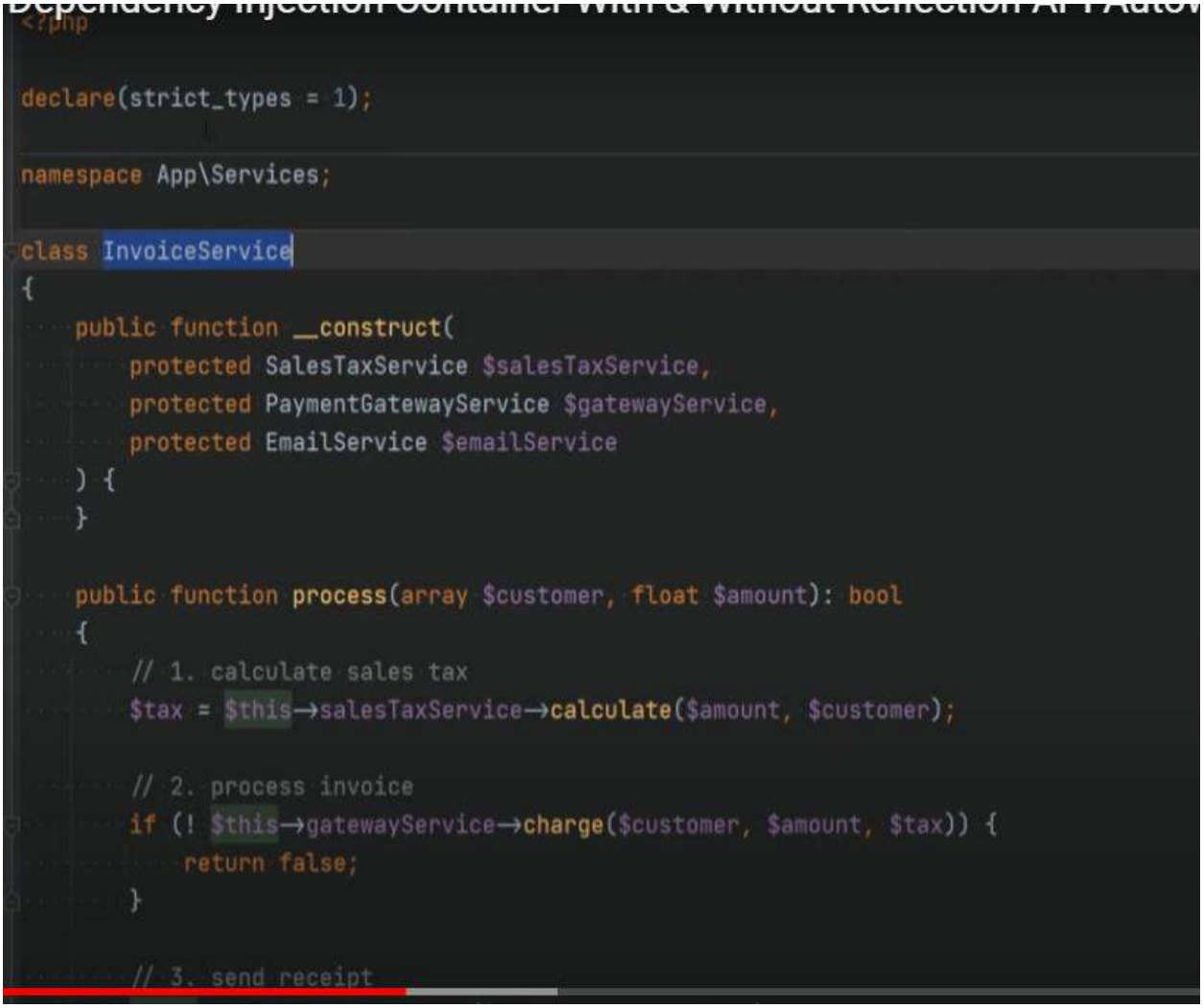
- Vì lớp "app" chính là lớp chính khởi động ứng dụng, nên sẽ tạo một thuộc tính tĩnh mới có tên "container" và thiết lập nó trong phương thức tĩnh.

Trong ví dụ này, chúng ta sẽ tạo ra một container và thêm các mục vào container trong hàm tạo của lớp "app." Tuy nhiên, bạn cũng có thể tạo một phương thức riêng để thêm các mục vào container, nếu bạn muốn tạo sự sắp xếp tốt hơn.



```php
namespace App;

use App\Exceptions\RouteNotFoundException;

class App
{
    private static DB $db;
    private static Container $container;

    public function __construct(protected Router $router, protected array $request, protected Config $config)
    {
        static::$db = new DB($config->db ?? []);
        static::$container = new Container();


    }

    public static function db(): DB
    {
        return static::$db;
    }

    public function run()
    {
        try {
```

- Ở bài học trước, chúng ta đã xem xét lớp "InvoiceService" làm ví dụ. Lớp này có ba phụ thuộc trong hàm tạo. Để có thể sau này lấy đối tượng "InvoiceService" từ container, chúng ta cần đăng ký hoặc thêm "InvoiceService" vào container.



- Đăng ký "InvoiceService" trong container bằng cách sử dụng phương thức "set."
- Chúng ta đã đăng ký "InvoiceService" trong container và sử dụng một hàm gọi (callback) để xác định cách tạo đối tượng "InvoiceService." Callback này hoạt động như một hàm giải quyết (resolver) và cho biết cho container cách tạo đối tượng "InvoiceService." Chúng ta đã lấy các phụ thuộc của "InvoiceService" từ container và sau đó tạo một đối tượng "InvoiceService" mới bằng cách truyền các phụ thuộc vào hàm tạo của nó.

```php
public function __construct(protected Router $router, protected array $request, protected Config $config)
{
    static::$db = new DB($config->db ?? []);
    static::$container = new Container();

    static::$container->set(InvoiceService::class, function(Container $c) {
        return new InvoiceService($c->get(SalesTaxService::class), $c->get(PaymentGatewayService::class), $c->get(EmailService::class));
    });
}
```

- Chúng ta đã đăng ký "InvoiceService" trong container và sử dụng một hàm gọi (callback) để xác định cách tạo đối tượng "InvoiceService." Để làm điều này dễ đọc hơn, chúng ta sẽ sắp xếp lại đoạn mã. Điều này có nghĩa là chúng ta cần thêm các mục này vào container.

```php
class App
{
    private static DB        $db;
    private static Container $container;

    public function __construct(protected Router $router, protected array $request, protected Config $config)
    {
        static::$db        = new DB($config->db ?? []);
        static::$container = new Container();

        static::$container->set(
            InvoiceService::class,
            function (Container $c) {
                return new InvoiceService(
                    $c->get(SalesTaxService::class),
                    $c->get(PaymentGatewayService::class),
                    $c->get(EmailService::class)
                );
            }
        );
    }
```

- Bên dưới là phiên bản dịch cho phần bạn đã nêu:

```php
private static DB        $db;
private static Container $container;

public function __construct(protected Router $router, protected array $request, protected Config $config)
{
    static::$db        = new DB($config->db ?? []);
    static::$container = new Container();

    static::$container->set(
        InvoiceService::class,
        function (Container $c) {
            return new InvoiceService(
                $c->get(SalesTaxService::class),
                $c->get(PaymentGatewayService::class),
                $c->get(EmailService::class)
            );
        }
    );

    static::$container->set(SalesTaxService::class, fn() => new SalesTaxService());
    static::$container->set(PaymentGatewayService::class, fn() => new PaymentGatewayService());
    static::$container->set(EmailService::class, fn() => new EmailService());
}
```

- Trong đoạn mã trên, chúng ta đã sử dụng hàm mũi tên (arrow function) để đăng ký các dịch vụ ("SalesTaxService," "PaymentGatewayService," và "EmailService") trong container. Mỗi callback function đơn giản là trả về một phiên bản mới của lớp tương ứng.
- Tuy nhiên, điều này không trông tốt bởi vì nó tạo ra sự lặp lại trong việc đăng ký các dịch vụ.
- Vậy bạn muốn lấy một phiên bản của lớp "InvoiceService" từ controller của bạn. Dưới đây là cách bạn có thể thực hiện việc này trong controller:

```php
<?php

declare(strict_types = 1);

namespace App\Controllers;

use App\View;

class HomeController
{
    public function index(): View
    {

        return View::make('index');
    }
}
```

- Cách chúng ta truy cập container là thông qua thuộc tính này đang được đánh dấu là riêng tư (private) hiện tại. Vậy hãy làm cho nó trở thành public tạm thời vì như tôi đã đề cập trước đó, chúng ta sẽ loại bỏ nó sau này, vì vậy hiện tại việc để nó public không sao.

```php
class App
{
    private static DB        $db;
    public static Container $container;

    public function __construct(protected Router $router, protected array $request, protected Config $config)
    {
        static::$db        = new DB($config->db ?? []);
```

- Nếu muốn sử dụng container để lấy một phiên bản của lớp "InvoiceService" và sau đó gọi phương thức "process" trên nó, dưới đây là có thể thực hiện điều này:

```php
declare(strict_types = 1);

namespace App\Controllers;

use App\App;
use App\Services\InvoiceService;
use App\View;

class HomeController
{
    public function index(): View
    {
        App::$container->get(InvoiceService::class)->process([], 25);

        return View::make('index');
    }
}
```

- Nếu muốn thêm một câu lệnh echo vào phương thức "process" trong lớp "InvoiceService" để kiểm tra liệu phương thức này có được gọi hay không, có thể thực hiện như sau:

```php
public function process(array $customer, float $amount): bool
{
    // 1. calculate sales tax
    $tax = $this->salesTaxService->calculate($amount, $customer);

    // 2. process invoice
    if (! $this->gatewayService->charge($customer, $amount, $tax)) {
        return false;
    }

    // 3. send receipt
    $this->emailService->send($customer, 'receipt');

    echo 'Invoice has been processed<br />';

    return true;
}
```

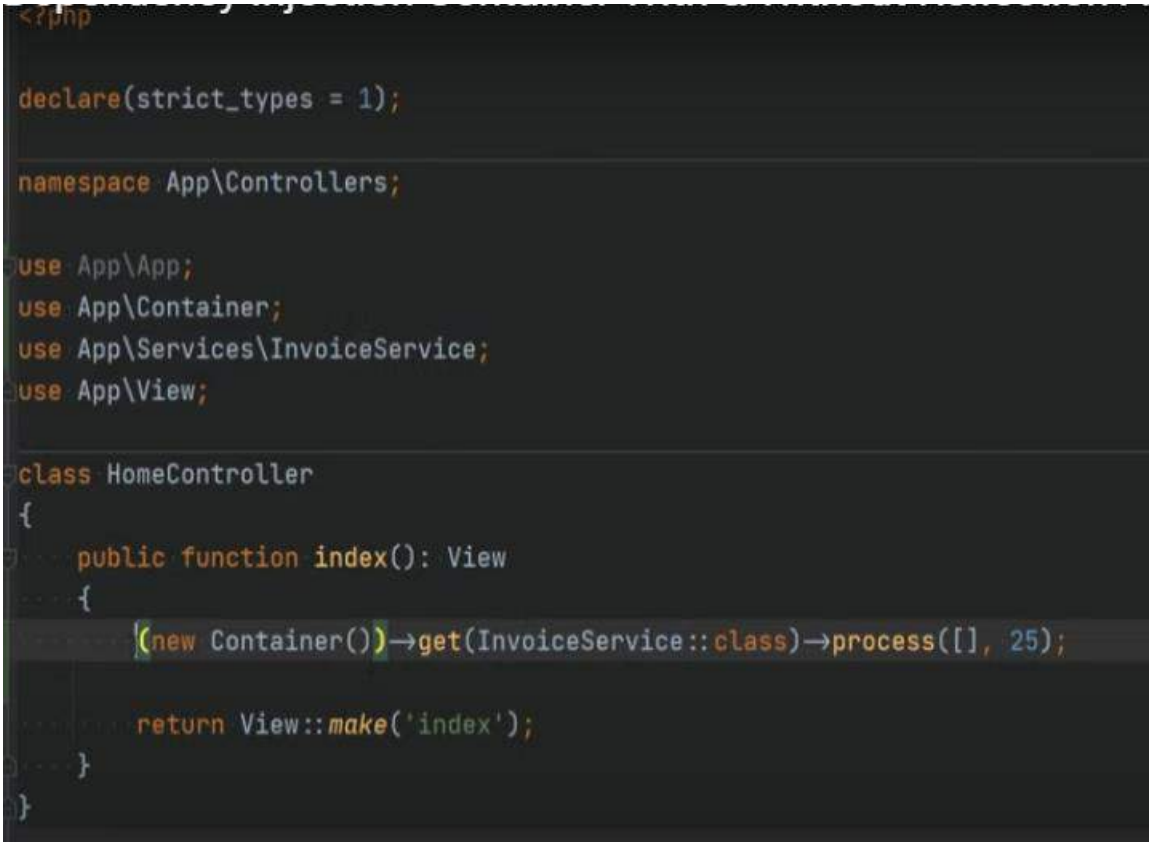
- Nếu bạn muốn tạm thời tắt các hàm sleep để không phải chờ ba giây trong quá trình thử nghiệm, có thể làm như sau:

```php
<?php

declare(strict_types = 1);

namespace App\Services;

class SalesTaxService
{
    public function calculate(float $amount, array $customer): float
    {
        sleep(1);

        return $amount * 6.5 / 100;
    }
}
```

- Hãy quay lại controller và mở trình duyệt để kiểm tra thực tế và chắc chắn rằng chúng ta đã thực hiện điều này:

```php
public function __construct(protected Router $router, protected array $request, protected Config $config)
{
    static::$db         = new DB($config->db ?? []);
}

public static function db(): DB
{
    return static::$db;
}
```

- Và chúng ta cũng có thể loại bỏ dòng mã sau từ HomeController:
- Chúng ta đã thay thế việc truy cập container tĩnh bằng cách sử dụng một hàm hoặc phương thức factory để lấy instance của container và sau đó lấy InvoiceService từ container.



```php
<?php

declare(strict_types = 1);

namespace App\Controllers;

use App\App;
use App\Container;
use App\Services\InvoiceService;
use App\View;

class HomeController
{
    public function index(): View
    {
        (new Container())->get(InvoiceService::class)->process([], 25);

        return View::make('index');
    }
}
```

- Để triển khai auto-wiring, cần làm như sau:

1. Kiểm tra lớp: Đầu tiên, chúng ta cần kiểm tra lớp mà chúng ta đang cố gắng lấy từ container.
2. Kiểm tra constructor: Sau đó, chúng ta cần kiểm tra constructor của lớp.
3. Kiểm tra tham số của constructor: Chúng ta cần xem xét các tham số của constructor, tức là các phụ thuộc.
4. Giải quyết tham số của constructor: Nếu tham số của constructor là một lớp, chúng ta cần thử giải quyết lớp đó bằng container.
5. Hãy bắt đầu với bước 1, kiểm tra lớp:
6. Trước khi chuyển sang bước 2, hãy kiểm tra xem lớp có thể được khởi tạo không:

- Hãy nhớ rằng chúng ta cần xử lý các trường hợp đặc biệt, như khi lớp là một giao diện hoặc lớp trừu tượng, nhưng chúng ta sẽ thực hiện điều này sau.

```php
public function get(string $id)
{
    if ($this->has($id)) {
        $entry = $this->entries[$id];

        return $entry($this);
    }

    return $this->resolve($id);
}
```

- Để triển khai bước 2 (kiểm tra constructor), cần sử dụng đối tượng ReflectionClass đã tạo ở bước 1 để lấy thông tin về constructor của lớp. có thể làm như sau:

```php
public function resolve(string $id)
{
    // 1. Inspect the class that we are trying to get from the container
    $reflectionClass = new \ReflectionClass($id);

    if (! $reflectionClass->isInstantiable()) {
        throw new ContainerException('Class "' . $id . '" is not instantiable');
    }

    // 2. Inspect the constructor of the class
    $constructor = $reflectionClass->getConstructor();

    // 3. Inspect the constructor parameters (dependencies)

    // 4. If the constructor parameter is a class then try to resolve that class using the container
}
```

- Nếu một tham số là kiểu lớp, có thể thực hiện bước 4 (giải quyết tham số của constructor) để tiếp tục quá trình triển khai auto-wiring.

- Nếu lớp không có constructor (tức là không có tham số), có nghĩa rằng không có phụ thuộc và chúng ta có thể trực tiếp trả về một phiên bản mới của lớp đó. Bạn đã đề xuất một cách tiếp cận tốt khi không cần tạo instance qua phương thức new của lớp. Sử dụng return new $classToResolve; là cách đơn giản và rõ ràng hơn.
- Nếu constructor không có tham số, chúng ta không cần phải tiếp tục kiểm tra và giải quyết bước 4, vì không có phụ thuộc nào để giải quyết.
- Bước này đã được thể hiện rất rõ trong mã của bạn:

```php
public function resolve(string $id)
{
    // 1. Inspect the class that we are trying to get from the container
    $reflectionClass = new \ReflectionClass($id);

    if (! $reflectionClass->isInstantiable()) {
        throw new ContainerException('Class "' . $id . '" is not instantiable');
    }

    // 2. Inspect the constructor of the class
    $constructor = $reflectionClass->getConstructor();

    if (! $constructor) {
        return new $id;
    }

    // 3. Inspect the constructor parameters (dependencies)
    $parameters = $constructor->getParameters();

    // 4. If the constructor parameter is a class then try to resolve that class using the container
}
```

- Nếu một lớp có constructor mà không có tham số, thì nó không có phụ thuộc và có thể tạo một phiên bản mới của lớp đó bằng cách sử dụng new và trả về nó ngay lập tức. Điều này rất dễ dàng và không cần phải giải quyết phụ thuộc.

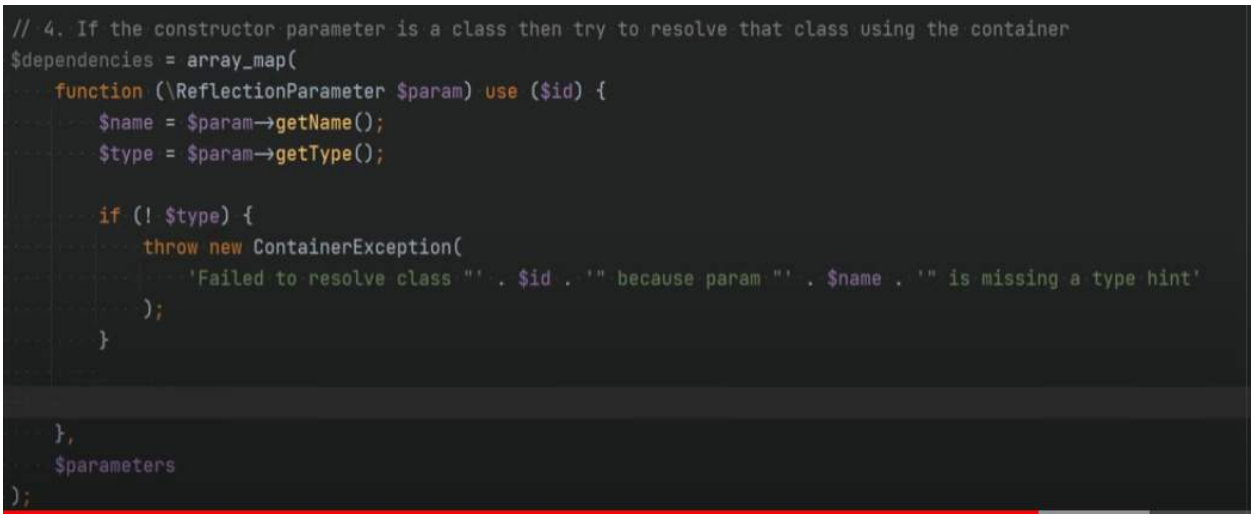
```php
public function __construct(
    protected SalesTaxService $salesTaxService,
    protected PaymentGatewayService $gatewayService,
    protected EmailService $emailService
) {
```

- Việc tách biệt các phần trong mã giúp làm cho mã dễ đọc và hiểu hơn. Tiếp tục với các bước tiếp theo của bạn trong việc triển khai dependency injection container.

```php
public function resolve(string $id)
{
    // 1. Inspect the class that we are trying to get from the container
    $reflectionClass = new \ReflectionClass($id);

    if (! $reflectionClass->isInstantiable()) {
        throw new ContainerException('Class "' . $id . '" is not instantiable');
    }

    // 2. Inspect the constructor of the class
    $constructor = $reflectionClass->getConstructor();

    if (! $constructor) {
        return new $id;
    }

    // 3. Inspect the constructor parameters (dependencies)
    $parameters = $constructor->getParameters();

    if (! $parameters) {
        return new $id;
    }
```

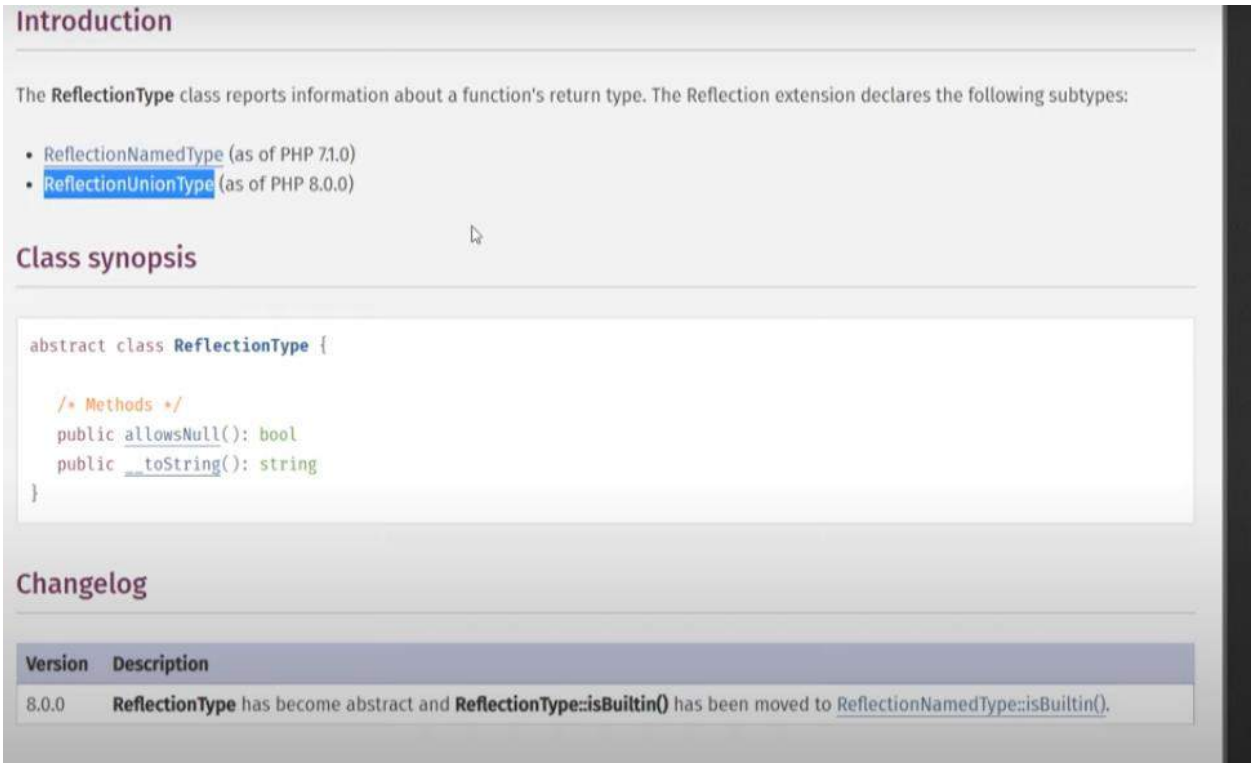
- Bây giờ hãy xem xét hàm getParameters để xem nó trả về gì. Chúng ta có thể thấy nó trả về một mảng các đối tượng reflection parameter. Do đó, nếu không có tham số nào, nó sẽ trả về một mảng trống, và đó là nơi chúng ta đang xử lý nó ở đây. Nếu nó không trả về một mảng trống, thì chúng ta biết rằng có một số tham số, và chúng ta cần xây dựng phần xử lý nằm trong bước thứ tư. Vì phần của bước thứ tư, chúng ta cần kiểm tra xem tham số của hàm tạo có phải là một class không, và sau đó cố gắng giải quyết class đó bằng cách sử dụng chính container. Đây chính là bước đệ quy ở đây, và chúng ta cần thực hiện điều này cho từng tham số. Bước thứ tư là nơi chúng ta sẽ xây dựng các phần tử trong dependencies.

```php
// 4. If the constructor parameter is a class then try to resolve that class using the container
$dependencies = array_map(
    function (\ReflectionParameter $param) {
        $name = $param->getName();
        $type = $param->getType();

    },
    $parameters
);
```



```php
// 4. If the constructor parameter is a class then try to resolve that class using the container
$dependencies = array_map(
    function (\ReflectionParameter $param) use ($id) {
        $name = $param->getName();
        $type = $param->getType();

        if (! $type) {
            throw new ContainerException(
                'Failed to resolve class "' . $id . '" because param "' . $name . '" is missing a type hint'
            );
        }

    },
    $parameters
);
```

- Như đã đề cập, trong PHP 8, reflection type đã trở thành một lớp trừu tượng và bạn không thể sử dụng isBuiltin() để xác định xem kiểu có phải là một kiểu dữ liệu nguyên thủy hay không. Thêm vào đó, union types (kiểu hợp) cũng đã được hỗ trợ, và chúng ta không muốn giải quyết chúng tự động trong trường hợp này.
- Để xử lý điều này, bạn có thể thêm một kiểm tra để xác định xem kiểu dữ liệu có phải là một kiểu dữ liệu nguyên thủy hoặc kiểu hợp không. Nếu là kiểu dữ liệu nguyên thủy hoặc kiểu hợp, bạn có thể ném một ngoại lệ hoặc xử lý nó theo cách phù hợp cho ứng dụng của bạn.
- Cụ thể, bạn có thể kiểm tra kiểu của đối tượng ReflectionType để xem xét trường hợp của kiểu dữ liệu nguyên thủy và kiểu hợp. Sau đó, bạn có thể thực hiện xử lý tương ứng.



Introduction

The **ReflectionType** class reports information about a function's return type. The Reflection extension declares the following subtypes:

- ReflectionNamedType (as of PHP 7.1.0)
- ReflectionUnionType (as of PHP 8.0.0)

Class synopsis

```
abstract class ReflectionType {

   /* Methods */
   public allowsNull(): bool
   public __toString(): string
}
```

Changelog

Version	Description
8.0.0	**ReflectionType** has become abstract and **ReflectionType::isBuiltin()** has been moved to ReflectionNamedType::isBuiltin().

- Bây giờ, để thực hiện bước số 4, bạn cần thử giải quyết lớp nếu tham số của constructor là một lớp. Để làm điều này, bạn có thể sử dụng tên lớp từ kiểu tham số và thực hiện giải quyết trong container. Dưới đây là cách bạn có thể làm điều đó:

```
// 4. If the constructor parameter is a class then try to resolve that class using the container
$dependencies = array_map(
    function (\ReflectionParameter $param) use ($id) {
        $name = $param->getName();
        $type = $param->getType();

        if (! $type) {
            throw new ContainerException(
                'Failed to resolve class "' . $id . '" because param "' . $name . '" is missing a type hint'
            );
        }

        if ($type instanceof \ReflectionUnionType) {
            throw new ContainerException(
                'Failed to resolve class "' . $id . '" because of union type for param "' . $name . '"');
        }

        if ($type instanceof \ReflectionNamedType && ! $type->isBuiltin()) {

        }

        throw new ContainerException(
            'Failed to resolve class "' . $id . '" because invalid param "' . $name . '"');
    },
    $parameters
);
```

- Trong trường hợp này, $className là tên của lớp mà bạn đang cố gắng giải quyết. Bạn có thể sử dụng phương thức resolve để giải quyết lớp trong container và sau đó thêm kết quả vào mảng dependencies.

- Dưới đây là cách thực hiện điều đó:

```
if ($type instanceof \ReflectionNamedType && ! $type->isBuiltin()) {
    return $this->get($type->getName());
}
```

- Như đã đề cập, điều này sẽ cho phép bạn giải quyết các phụ thuộc một cách đệ quy. Nếu lớp SalesTaxService có các phụ thuộc của riêng nó, chúng cũng sẽ được giải quyết theo cùng một cách, giúp xây dựng cây phụ thuộc đầy đủ.

```
return $reflectionClass->newInstanceArgs($dependencies);
```

- Đúng vậy, bạn đã hiểu rất đúng. Điều quan trọng là bạn đã triển khai một DI container đơn giản nhưng hoàn toàn hoạt động để hiểu cách nó hoạt động bên trong.