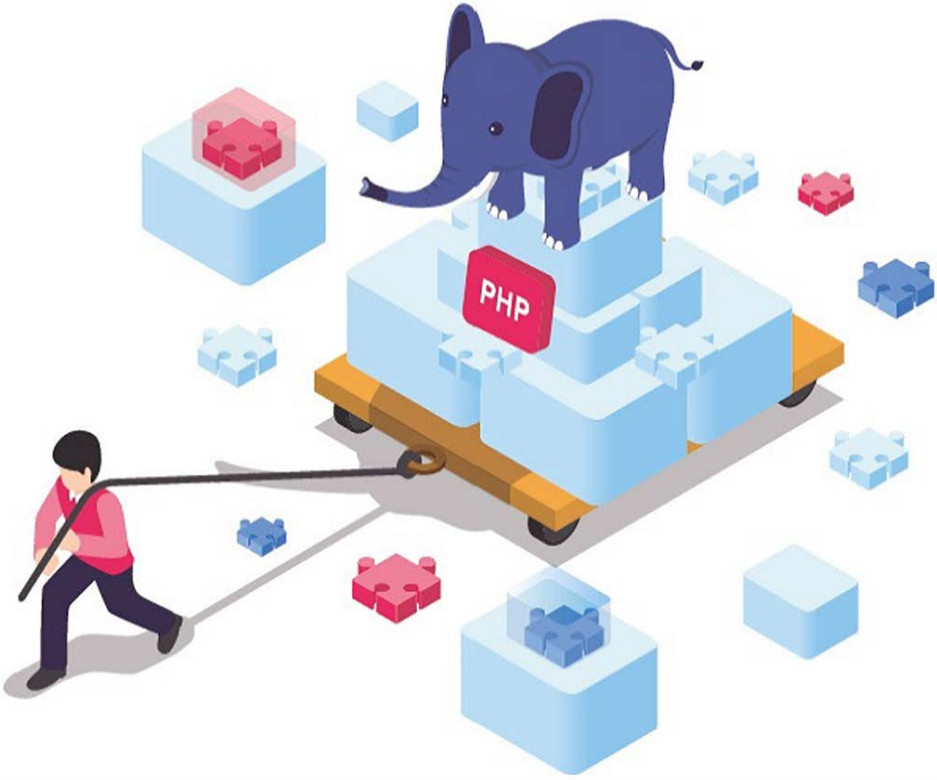
PSRs

Boas práticas de programação com PHP



Casa do Código

JHONES S. CLEMENTINO

ISBN

Impresso e PDF: 978-85-7254-030-8

EPUB: 978-85-7254-031-5

MOBI: 978-85-7254-032-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Primeiramente agradeço a Deus por ter me concedido forças nos momentos mais difíceis em que pensei que não iria superar algumas dificuldades da minha vida. Agradeço a minha esposa que me incentivou muito desde o começo até o término desta obra. Também faço agradecimentos aos meus pais por terem me proporcionado o melhor que puderam e também por terem feito de tudo para que eu tivesse uma boa educação e pudesse avançar na escalada da vida.

Por fim, agradeço a todas as pessoas com quem trabalhei em todos esses anos favorecendo e muito para a troca de conhecimentos e aprendizagem de novas tecnologias e conceitos.

SOBRE O AUTOR

Eu me chamo Jhones dos Santos Clementino, sou apaixonado por programação desde os 17 anos, quando descobri que os softwares, games e sites eram desenvolvidos através de alguma linguagem de programação - essa descoberta mudou minha vida. Comecei a me interessar por esses assuntos cada vez mais porque achava incrível uma sequência de código fazer algo tão útil e interessante como os jogos, por exemplo. Isso é fascinante!

Sou formado em Ciência da Computação pela Universidade Paulista - UNIP e trabalho com desenvolvimento de sistemas WEB desde 2009 quando ocorreu meu primeiro contato com o PHP. Desde aquela época fui me dedicando a aprender mais e mais com cursos online, tutoriais, livros e apostilas. Meu foco tem sido a Web porque são tecnologias que estão em constante evolução.

Também sou o autor do livro: *Zend Expressive e PHP 7: Uma união poderosa para a criação de APIs*, publicado pela editora Casa do Código.

Quando possuo um tempo livre gosto de fazer alguma coisa que tire a minha atenção do mundo virtual por algum tempo, então gosto de sair com a minha mulher, desenhar e até mesmo cantar (vamos deixar isso para uma outra hora, OK? Rs). Bom, meu amigo, esse é um resumão de quem sou eu.

PREFÁCIO

Hoje em dia, o número de aplicações disponíveis é absurdamente grande, e assim como é o número de aplicações, também é a quantidade de códigos.

Porém, será que esses códigos são bem escritos? Será que qualquer desenvolvedor consegue dar manutenção sem dificuldades? A resposta para essas perguntas é que depende muito da equipe que desenvolveu, depende muito das regras de cada linguagem de programação.

Não existe no mundo um meio de padronizar como um desenvolvedor deve pensar, para que os códigos sejam escritos de forma padrão tanto em lógica, quanto em regras de formatação.

Em PHP não é diferente, basta você trabalhar em alguma aplicação legada que já está em funcionamento desde os primórdios de uma empresa, que você verá o quão assombroso é o código em sua frente.

Neste livro, vamos abordar as boas práticas de programação com o PHP utilizando as PSRs (*PHP Standards Recommendations*), e veremos os mais variados tipos de conceitos e como aplicá-los em nosso código.

Este livro é indicado para todos os tipos de desenvolvedores desde os estagiários, estudantes, ao mais experientes, portanto, não é necessário ter conhecimento avançado em PHP, mas é importante conhecer o básico da linguagem e de Programação Orientada a Objetos. O conhecimento básico de ambos é

indispensável pois eles serão utilizados com frequência.

Caso você possua dúvidas, críticas, sugestões ou correções, poderá entrar em contato através do e-mail: **jhones.developer@gmail.com** ou através do LinkedIn: https://www.linkedin.com/in/jhones-dos-santos-clementino-91a90256/.

No decorrer deste livro vamos abordar diversas PSRs, desde as que cuidam da formatação do código até as que cuidam de maiores complexidades como troca de mensagens HTTP, logs, cache etc.

Então, sem desanimar, siga em frente e vamos nessa!

Sumário

Capítulo 1

INTRODUÇÃO

Se você já é um desenvolvedor que possui um certo nível de experiência, com certeza já se deparou com códigos mal escritos, mal indentados, códigos horrendos que fazem até mesmo você questionar a sua própria sanidade mental e ficar se perguntando: como é que isso está funcionando ainda? Funciona mesmo? Não pode ser... Pois é, meu amigo, eu digo, pode sim, e bem-vindo ao mundo digital onde certezas e incertezas tomam os seus pensamentos durante o processo de reconhecimento de códigos criados por mentes maquiavélicas.

Se você já passou horas e até mesmo dias tentando entender o impossível, então você já sabe o quanto é difícil dar manutenção em um código desse tipo, totalmente fora de padrão, sem ter explicação lógica para o monstro à sua frente. Acontece que em PHP até um tempo atrás não havia um grupo focado em definir padrões e regras que deveriam ser seguidas para que o código ficasse o mais legível e padronizado possível. Felizmente, em 2009 ocorreu o surgimento de um abençoado grupo chamado **PHP-FIG (Framework Interop Group)**.

O PHP-FIG é um grupo focado em definir padrões para o PHP. Em outras palavras, o grupo é responsável por criar padrões e regras em comum que possam ser utilizados em qualquer tipo de

projeto em PHP. Regras de carregamento de classes (`autoloader`), de formatação do código-fonte, regras de log, dentre muitas outras que veremos mais adiante neste livro.

Foi então criado o que chamamos de **PSR** (*PHP Standard Recommendation* ou Padrão Recomendado para PHP, em português), que são especificações que foram definidas pelos membros que fazem parte do PHP-FIG. São membros como Zend Framework, Symfony, Slim, Composer e muitos outros que participam de projetos diferentes para chegar a soluções concisas que ajudam todos os desenvolvedores PHP - isso sem dúvidas!

É importante conhecer que o PHP-FIG define algumas palavras-chaves dentro das PSRs que servem como uma identificação no projeto. São elas:

- **MUST** - DEVE
- **MUST NOT** - NÃO DEVE
- **REQUIRED** - OBRIGATÓRIO
- **SHALL** - TEM QUE
- **SHALL NOT** - NÃO TEM QUE
- **SHOULD** - DEVERIA
- **SHOULD NOT** - NÃO DEVERIA
- **RECOMMENDED** - RECOMENDADO
- **MAY** - PODE
- **OPTIONAL** - OPCIONAL

Não se preocupe se você não estiver entendendo para que servem essas palavras-chaves. Quando entrarmos em cada PSR você entenderá.

Agora que você já sabe que existe um grupo focado em criar

padrões para serem seguidos em PHP, está na hora de conhecermos todas as PSRs disponíveis até o momento da escrita deste livro.

Capítulo 2

PSR-1 (BASIC CODING STANDARD)

Essa PSR tem como objetivo definir alguns elementos padrões de codificação que devem garantir um maior nível de interoperabilidade entre os códigos PHP.

> **O que é interoperabilidade?**
>
> Interoperabilidade é a capacidade de um sistema ou aplicação comunicar-se com outros sistemas ou aplicações de maneira eficiente, alcançando um maior nível de integridade dos dados.

Garantir que a sua aplicação seja interoperável traz consigo uma maior integridade dos dados, e além disso, a sua aplicação torna-se mais eficiente quando precisar se comunicar com outras aplicações.

Já aplicações que não possuem um alto nível de interoperabilidade tendem a ter dificuldades em se comunicar com outras, e dentro de um ambiente corporativo isso pode levar a

drásticas consequências, como refatorar a aplicação de maneira que possa atender às necessidades da empresa ou até mesmo iniciar a criação de uma nova aplicação mais eficiente.

Portanto, se a aplicação for interopeŕavel, você não deverá ter dor de cabeça no momento em que precisar realizar a comunicação com outros sistemas.

2.1 REGRAS

Confira nesta seção o conjunto de todas as regras e características dessa PSR.

Regra 1

Arquivos PHP DEVEM usar apenas as tags `<?php` e `<?=`. Isso significa que qualquer outro tipo de tag, como a ASP tag `(<%)` ou a tag de script PHP, não deve ser utilizado. Em muitos servidores, as *short open tags* estão desabilitadas por padrão e isso pode ocasionar muita dor de cabeça, então por convenção utilize somente as tags `<?php` ou `<?=`.

Regra 2

Arquivos PHP DEVEM usar apenas UTF-8 sem BOM (*Byte Order Mark*). Sabe quando você abre um arquivo que não está utilizando esse padrão e aparece um monte de código estranho nele? Pois bem, isso acontece porque o arquivo provavelmente está usando uma codificação com BOM, por exemplo: ISO-8859-1 ou UTF-8 com BOM.

Foi definido que o padrão a ser seguido e utilizado é a UTF-8

sem BOM, desse modo, os problemas de incompatibilidade de codificação acabam. Vale ressaltar que maioria das IDEs já vem configurada para o padrão UTF-8 sem BOM.

Regra 3

Arquivos PHP DEVERIAM declarar (classes, métodos, constantes, atributos etc.) ou causar efeitos colaterais (gerar outputs, realizar modificações no `php.ini` etc.), mas NÃO DEVERIAM fazer as duas coisas.

Imagine um arquivo PHP que possui a declaração de uma classe, mas também possui trechos de código que alteram a configuração do `php.ini` e que ainda possui exibição de conteúdo na tela. Já sabe a bagunça e a zona que é: tudo desorganizado, misturando as responsabilidades em uma verdadeira sopa de lógica mal definida. E isso resulta em um tempo considerável para fazer a correção e a separação das responsabilidades.

Se seu arquivo for de declaração de uma classe ou interface, então respeite. Faça apenas isso, nada de exibir conteúdo ou alterar configurações do `php.ini` antes da declaração da classe ou após a declaração. Para fazer isso, crie um arquivo que seja responsável por fazer essas tarefas, ok?

Veja o exemplo a seguir para melhor entendimento sobre um arquivo que faz exatamente ambas as coisas, o que DEVE ser evitado:

```
<?php
//Alterando a configuração do php.ini
ini_set('error_reporting', E_ALL);
```

```php
//Incluindo um arquivo externo
include "arquivo.php";

//Gerando uma saída na tela
echo "<html>\n";

//Declarando uma classe
class Usuario
{
}
```

Como você pôde ver, é um tipo de exemplo que - para ser bem sincero - é muito mais comum do que se pensa. Basta você trabalhar com qualquer tipo de sistema legado em PHP que você já encontrará algumas assombrações dessas, então, lembre-se: evite colocar em um mesmo arquivo declarações e outputs como o exemplo citado.

Agora veremos um exemplo que seria o correto, um arquivo apenas com a declaração:

```php
<?php
//Declarando uma classe
class Usuario
{
    private $nome;

    public function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

Note que nesse exemplo não há nenhuma geração de output no arquivo de declaração. Ele faz exatamente a declaração da classe, do método e do atributo, nada além disso.

Regra 4

Namespaces e classes DEVEM seguir o padrão descrito na PSR-0 (depreciada) e PSR-4 (substituta da PSR-0). Para que se obtenha um `autoloading` compreensível e eficaz é necessário seguir a mesma estrutura de diretórios no namespace na declaração dos arquivos.

> **O QUE É AUTOLOADING?**
>
> É a maneira de realizar o carregamento automático de arquivos como classes, interfaces, traits etc., sem que haja a necessidade de o desenvolvedor vincular esses arquivos explicitamente no código.

O autoloading de arquivos é considerado compreensível e eficaz quando realizado da maneira correta, ou seja, da maneira que é estipulada por esta PSR.

Para entendermos melhor, vamos ao exemplo a seguir:

```
|-src
  |-App
    |-Controller
      |-HomeController.php
```

Note que temos uma estrutura de diretórios bem simples. Com a utilização de namespace, o exemplo ficaria conforme a seguir:

```
<?php
namespace App\Controller;

class HomeController
```

```
{
}
```

Temos uma classe `HomeController` que está dentro do diretório `App` e consequentemente dentro do subdiretório `Controller`, logo, o namespace dessa classe é `App\Controller`. Não é uma coisa difícil de se aprender, é apenas uma questão habitual.

A utilização de namespaces facilita e muito, pois não precisamos ficar dando vários `require()` no arquivo, apesar de que por trás da lógica de `autoloading` ele faz um `require` para o arquivo original.

Então, em vez de fazer `require __DIR__ '/src/App/Controller/HomeController.php'`, utilize o padrão considerado correto, que é através da utilização de namespaces.

Regra 5

O nome de arquivos como classes, interfaces e traits DEVEM seguir o padrão `StudlyCaps`. Isso significa que a primeira letra de cada palavra deve ser escrita com letra maiúscula, como é o caso do nosso exemplo `HomeController`.

Regra 6

Constantes DEVEM ser declaradas com todas as letras em maiúsculo e separadas por *underscores* (_). Nada de declarar uma constante com letras em minúsculo, ou `camelCase`, ou `snake_case`. O único modo permitido é a utilização de todas as letras em maiúsculo.

Para facilitar o entendimento, vamos ao exemplo a seguir do que é considerada a maneira correta na declaração de constantes:

```
<?php
const LOG_CRITICAL = 'critical';
```

Nada demais, não é mesmo? E o jeito considerado incorreto? Vamos verificar a seguir:

```
<?php
const logCritial = 'critical';
const log_info = 'info';
```

Este último exemplo mostra uma das formas incorretas de declarar uma constante em PHP, então evite declará-las assim.

Regra 7

As propriedades definidas DEVERÃO ser aplicadas de forma consistente dentro de um escopo razoável, podendo ser em nível de pacote, em nível de fornecedor, de classe ou de método. Não importa a convenção de nomenclatura utilizada (`$StudlyCaps` , `$camelCase` , `$snake_case`) desde que respeitadas as regras anteriormente descritas.

Regra 8

O nome dos métodos DEVE ser declarado na convenção de nomenclatura `camelCase` . Essa regra também não é nada demais, basta atentar-se em escrever o nome dos métodos neste padrão. Para facilitar o entendimento, vamos ao exemplo a seguir que demonstra a forma correta de definir o nome de um método:

```
<?php

public function getNomeUsuario()
```

```
{
}
```

Repare que o nome do método foi definido utilizando a convenção de nomenclatura `camelCase` . Agora vamos ver as formas consideradas incorretas ao declarar o nome de um método:

```
<?php

//Declaração em snake_case
public function get_nome_usuario()
{
}

//Declaração em StudlyCaps
public function GetNomeUsuario()
{
}
```

Conclusão

Chegamos ao final da PSR-1, que mostra para nós algumas regras que devem ser seguidas para que o nosso código fique mais legível e adequado para o desenvolvimento. Não é uma PSR complexa, mas vale a pena atentar-se às regras dela, para que você possa aplicá-las com mais precisão e eficácia em seus códigos.

No próximo capítulo falaremos sobre a PSR-2 que trata das regras de legibilidade do código-fonte, então vamos nessa!

Capítulo 3

PSR-2 (CODING STYLE GUIDE) - PARTE I

Está cansado de olhar um código e ter que ficar perdendo tempo fazendo a indentação e colocando tudo em seu devido lugar para melhorar o seu entendimento? Com essa PSR seus problemas podem acabar. A PSR-2 surgiu com o objetivo de tornar a escrita e a legibilidade do código mais simples e fácil para os desenvolvedores.

Veremos na seção a seguir quais são as regras, quais são os conceitos e como aplicá-los em seu projeto para que se obtenha um código mais bonito e legível.

Informação importante

A PSR-2 foi **descontinuada** devido a inúmeras funcionalidades que foram surgindo no PHP; mas não se preocupe, a PSR substituta é a **PSR-12** e a veremos mais adiante neste livro. Você pode continuar lendo a **PSR-2** sem problemas, pois muitas coisas presentes nela foram reaproveitadas/estendidas na **PSR-12** e você poderá conhecer as diferenças entre elas e as peculiaridades específicas para o PHP 7.

3.1 REGRAS

Confira nesta seção algumas regras e características dessa PSR.

Regra 1

Os códigos PHP DEVEM seguir os padrões estabelecidos na PSR-1 (*Basic Code Standard*). Por ser uma extensão da PSR-1, obviamente deve-se seguir o padrão apresentado nela, antes de se aplicar a PSR-2.

Regra 2

Arquivos PHP DEVEM possuir o padrão UNIX de terminação de linha. Você sabe quais são as terminações de linhas disponíveis? Caso não, veja a seguir:

- **CRLF**: retorno de carro + avanço de linha, Unicode

caracteres D + 000A 000
- **LF**: alimentação de linha, caracteres Unicode 000A
- **NEL**: próxima linha, o caractere Unicode 0085
- **LS**: separador de linha, caracteres Unicode 2028
- **PS**: separador de parágrafo, caractere Unicode 2029

O padrão UNIX adotado pela PSR-2 é o LF. Geralmente as IDEs já vêm configuradas para o padrão UNIX, mas isso pode ser alterado facilmente, dependendo da IDE escolhida.

Regra 3

Códigos PHP DEVEM possuir indentação com 4 espaços e não com TABs. Não é uma regra complexa, basta atentar-se e evitar o costume de utilizar TABs. Algumas IDEs já vêm com o padrão de 4 espaços ao teclar o TAB, mas vale conferir para que seu código não saia das conformidades da PSR-2. Isso pode ser configurado sem muita dor de cabeça.

Regra 4

Arquivos contendo apenas PHP DEVEM ter a TAG de fechamento omitida. Pode parecer estranho no primeiro momento, mas sim, o seu script PHP não vai deixar de funcionar se a TAG de fechamento (`?>`) for omitida. A regra não se aplica aos arquivos contendo HTML + PHP. Para facilitar o entendimento, vamos ver o exemplo a seguir, que demonstra a forma correta de aplicar essa regra:

```
<?php
class Usuario
{
    private $nome;
```

```
    public function __construct($nome)
    {
        $this->nome = $nome;
    }
}
```

Perceba que a TAG de fechamento (`?>`) foi omitida e isso não afeta o funcionamento do código. Agora vamos analisar o exemplo em que a omissão da TAG ocasionará em erro no código:

```
<?php
$nome = 'Jhones S. Clementino';

<html>
    <head>
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá, <?php echo $nome</span>!</h1>
    </body>
</html>
```

Diferentemente do exemplo anterior, a omissão da TAG de fechamento do PHP (`?>`) ocasionará erro se após a declaração da variável `$nome` não houver a TAG de fechamento. Isso porque o arquivo não possui apenas código PHP.

Regra 5

Arquivos contendo apenas código PHP DEVEM possuir uma linha em branco ao seu final. Após toda a lógica aplicada, deverá existir uma linha em branco.

Regra 6

O comprimento máximo de uma linha de código DEVE ser de 120 caracteres. Verificadores de estilo automatizado DEVEM

avisar, mas NÃO DEVEM exibir erro.

As linhas de código NÃO DEVEM ter mais de 80 caracteres. Linhas de código mais longas DEVEM ser quebradas em várias linhas subsequentes de no máximo 80 caracteres.

Pode parecer estranha a explicação, mas faz todo o sentido. Em sistemas legados, principalmente, há linhas com mais de 1000 caracteres de comprimento - acredite, não é exagero! Já trabalhei em sistemas desse tipo e é complicado dar manutenção nesse tipo de código.

Portanto, atente-se para que a cada 80 caracteres você possa quebrar a linha de modo que fique legível; caso não consiga, você ainda possui o limite de 120 caracteres.

Não é um erro, mas se quiser seguir o padrão da PSR-2 e manter suas boas práticas é muito importante atentar-se a mais esta regra, ok?

Regra 7

Linhas que possuem conteúdo NÃO DEVEM possuir espaços em branco ao final de seu conteúdo.

Regra 8

Para facilitar a interpretação do código, linhas em branco PODEM ser adicionadas. Você pode adicionar linhas em branco para tornar a interpretação do código mais fácil. Vamos ver a seguir um breve exemplo da aplicação dessa regra:

```php
<?php

class Usuario
{
    private $nome;

     private $idade;

     private $sexo;

    public function __construct($nome, $idade, $sexo)
    {
          $this->nome = $nome;
         $this->idade = $idade;

         if ($sexo == 'M') {
            $this->sexo = 'Masculino';
        } elseif ($sexo == 'F') {
            $this->sexo = 'Feminino';
        }
    }
}
```

Perceba como o código ficou legível com a aplicação das quebras de linha. Agora veja o exemplo a seguir em que não existe quebra de linha na escrita da lógica:

```php
<?php

class Usuario
{
    private $nome;
     private $idade;
     private $sexo;
    public function __construct($nome, $idade, $sexo)
    {
          $this->nome = $nome;
         $this->idade = $idade;
         if ($sexo == 'M') {
            $this->sexo = 'Masculino';
        } elseif ($sexo == 'F') {
            $this->sexo = 'Feminino';
        }
```

```
    }
}
```

O código funciona sem nenhum problema, mas perceba como fica um pouco estranho.

Regra 9

NÃO DEVE haver múltiplas declarações em uma única linha. Cada declaração de variável deve ser feita em uma nova linha. Definiu uma variável? Então pule uma linha para definir a próxima variável.

Vamos ver o exemplo a seguir que demonstra a maneira considerada correta ao declarar uma variável seguindo o padrão da PSR-2:

```
<?php
$nome = 'Jhones';
$sobrenome = 'S. Clementino';
```

Está correto, do jeito que tem que ser, uma declaração por linha. Agora veremos a maneira considerada incorreta:

```
<?php
$nome = 'Jhones'; $sobrenome = 'S. Clementino';
```

Este é um exemplo bem simples que não é considerado correto pela PSR-2, então lembre-se de declarar uma variável, constante, função etc. por linha.

Regra 10

As palavras-chaves e constantes `true`, `false` e `null` DEVEM ser escritas com letras minúsculas. Simplesmente use-as com letras minúsculas e não use variações como: `TRUE`, `FALSE`,

`NULL` ou `True`, `False`, `Null`, ou `TrUe`, `FaLsE`, `NuLl`.

Regra 11

Logo após a definição de um namespace, DEVE haver uma linha em branco. Quando você for definir um namespace, deve haver uma linha em branco antes de dar continuidade no código. Vamos a um exemplo para simplificar a explicação. O exemplo a seguir demonstra a maneira considerada correta:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    //Código
}
```

Perceba que logo após a definição do namespace uma linha em branco foi adicionada antes de declarar a classe `Usuario`. Agora veremos um exemplo da maneira considerada incorreta na definição do namespace:

```
<?php

namespace App\Entity\Usuario;
class Usuario
{
    //Código
}
```

Nesse exemplo não existe uma linha em branco adicionada após a definição do namespace, sendo considerado incorreto pela PSR-2. Então, sempre deixe uma linha em branco antes de prosseguir com o código.

Regra 12

Todas as declarações `use` existentes DEVEM estar após o namespace. Há momentos em que queremos importar um código que está contido em outra classe, e para incorporá-lo em nossa classe precisamos utilizar a declaração `use` .

Para entendermos melhor, vamos ao exemplo considerado correto de acordo com a PSR2:

```
<?php

namespace App\Controller\Usuario;

use App\Controller\AbstractController;

class UsuarioController extends AbstractController
{
    //Código
}
```

Perceba que após a declaração do namespace estou importando o controller `AbstractController` para dentro da minha classe. A declaração `use` está abaixo da definição do namespace da classe. Agora vamos analisar um exemplo em que a declaração `use` estará incorreta:

```
<?php

use App\Controller;

namespace App\Controller\Usuario;

class UsuarioController extends AbstractController
{
    //Código
}
```

Esse código não está correto e nem segue o padrão estabelecido

pela PSR-2, pois toda declaração `use` deve estar após a definição do namespace.

Regra 13

Apenas DEVE existir um `use` por declaração. Cada `use` deve possuir uma declaração. Vamos ao exemplo que simplifica a explicação demonstrando a maneira correta:

```
<?php

namespace App\Service\Usuario;

use App\Service\AbstractService;
use App\Entity\Usuario;

class UsuarioService extends AbstractService
{
    //Código
}
```

Como pode ser notado, para cada declaração foi utilizado um `use` em cada linha. Apenas atente-se no momento de utilizar o `use` . Agora vamos verificar o exemplo considerado incorreto na utilização do `use` :

```
<?php

namespace App\Service\Usuario;

use App\Service\AbstractService, App\Entity\Usuario;

class UsuarioService extends AbstractService
{
    //Código
}
```

Ter múltiplas declarações em uma única linha `use` não é um erro, então este código até vai funcionar. Entretanto seu código já

não estará em conformidade com a PSR-2.

Regra 14

As palavras-chaves `extends` e `implements` DEVEM ser declaradas na mesma linha que contém o nome da classe. Sempre que houver a necessidade de estender uma classe e/ou implementar uma interface, é necessário que ambas estejam na mesma linha que contém o nome da classe, pois também fazem parte da definição da classe.

Vamos ver o exemplo a seguir que demonstra a maneira correta de fazer isso:

```
<?php

namespace App\Service\Usuario;

use App\Service\ServiceInterface;
use App\Service\AbstractService;

class UsuarioService extends AbstractService implements ServiceIn
terface
{
    //Código
}
```

Nada demais até aqui. Como foi dito anteriormente, `extends` e `implements` fazem parte da definição de uma classe, então devem estar na mesma linha. Vamos analisar agora um exemplo incorreto:

```
<?php

namespace App\Service\Usuario;

use App\Service\ServiceInterface;
use App\Service\AbstractService;
```

```
class UsuarioService
extends AbstractService
implements ServiceInterface
{
    //Código
}
```

Esse código até funcionará, porém não estará em conformidade com a regra estabelecida pela PSR-2.

Regra 15

Quando houver a necessidade de múltiplos `implements` , eles PODEM ser divididos em múltiplas linhas, sendo que o primeiro item da lista DEVE estar na próxima linha e DEVE possuir somente uma interface por linha.

Se você ainda nunca viu ou ouviu falar em uma classe implementando múltiplas interfaces, saiba que não é incorreto e, sim, existem muitos projetos que, devido à complexidade, acabam implementando esse tipo de lógica.

Para melhor entendimento, vamos analisar o exemplo a seguir que demonstra a maneira considerada a correta pela PSR-2:

```
<?php

namespace App\Service\Usuario;

use App\Service\ServiceInterface;
use App\Filter\FilterInterface;
use App\Validator\ValidatorInterface;

class UsuarioService implements
    ServiceInterface,
    FilterInterface,
    ValidatorInterface
{
```

```
    //Código
}
```

O exemplo demonstra com clareza exatamente o que a PSR-2 define, ou seja, quando há múltiplas interfaces, cada interface deve estar em uma nova linha.

Agora vamos analisar o mesmo código, porém com as interfaces em uma única linha, sendo considerada a maneira incorreta:

```
<?php

namespace App\Service\Usuario;

use App\Service\ServiceInterface;
use App\Filter\FilterInterface;
use App\Validator\ValidatorInterface;

class UsuarioService implements ServiceInterface, FilterInterface
, ValidatorInterface
{
    //Código
}
```

Apesar de o código não estar em conformidade com a PSR-2 ele funcionará normalmente.

Regra 16

A visibilidade ou modificadores de acesso DEVEM ser definidos em todas as propriedades. Toda propriedade que for definida deve possuir o modificador de acesso definido antes do seu nome. Isso não é novidade, mas para quem não é acostumado a definir a visibilidade de uma propriedade, basta atentar-se sempre antes de criá-la.

Vamos analisar um exemplo em que as propriedades são definidas com os modificadores de acesso e que é a maneira considerada correta:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;
    private $idade;
    public $etnia;
    protected $dataDeNascimento;
}
```

Perceba que todas as propriedades que foram definidas possuem antes de seu nome o modificador de acesso, seja ele `private`, `protected` ou `public`. Agora vamos analisar o mesmo exemplo, porém sem os modificadores de acesso:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    $nome;
    $idade;
    $etnia;
    $dataDeNascimento;
}
```

Analisando o código do exemplo anterior, fica difícil de saber qual propriedade é privada, pública ou protegida.

Na verdade, quando o modificador de acesso não é definido na propriedade, um erro chamado **PHP Parse error** será lançado, pois dentro de uma classe obrigatoriamente as suas propriedades

devem possuir o modificador de acesso.

Regra 17

A palavra-chave `var` NÃO DEVE ser utilizada para a definição de propriedades da classe. No momento de definir a propriedade da classe, não utilize `var`, ok?

Regra 18

NÃO DEVE haver mais de uma propriedade por declaração. Se houver a necessidade de declarar mais de uma propriedade, então defina-as de forma separada, conforme mostrado no exemplo a seguir:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;
    private $idade;
    public $etnia;
    protected $dataDeNascimento;
}
```

Esse é o mesmo exemplo citado anteriormente. Como pode ser visto, cada declaração de propriedade está definida de forma separada. Agora veremos a maneira incorreta de se declarar as propriedades da classe:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
```

```
    private $nome, $idade, $etnia, $dataDeNascimento
}
```

Se você testar, o código funcionará perfeitamente. Mas lembre-se: o código não estará em conformidade com a PSR-2.

Regra 19

O nome das propriedades NÃO DEVE iniciar com `_` (*underscore*) para indicar a visibilidade `private` ou `protected`. Certamente você já deve ter visto em algum lugar que há programadores que definem as propriedades privadas ou protegidas da classe com um `_` como prefixo.

Isso não é necessário, uma vez que o PHP fornece os modificadores de acesso - basta você utilizá-los. Vamos analisar a maneira correta ao definir as propriedades privadas ou protegidas de uma classe:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;
    protected $idade;
}
```

Perceba que não há segredo algum, o nome das propriedades está limpo e claro como deve ser. Agora vamos analisar um exemplo considerado incorreto:

```
<?php

namespace App\Entity\Usuario;

class Usuario
```

```
{
    private $_nome;
    protected $_idade;
}
```

Nesse exemplo, o modificador de acesso já informa o tipo de visibilidade da propriedade tornando completamente inútil o uso do _ (underscore).

Regra 20

Todos os métodos DEVEM ter a visibilidade definida. Lembre-se de que a visibilidade torna o encapsulamento mais claro. Para facilitar, vamos ao exemplo a seguir:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    public function getNome()
    {
        return $this->nome;
    }

    public function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

Perceba que os métodos `getNome()` e `setNome($nome)` possuem o modificador de acesso `public` definido, indicando que eles possuem visibilidade pública.

O tipo de visibilidade vai depender do que será necessário

expor ou não para o resto da aplicação.

Agora vamos analisar o exemplo considerado incorreto e que não está em conformidade com a PSR-2:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    function getNome()
    {
        return $this->nome;
    }

    function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

É importante ressaltar que o código do exemplo funciona, mas não é válido para a regra da PSR-2.

Regra 21

Underscore (_) NÃO DEVE ser utilizado para indicar a visibilidade `private` ou `protected` do método. Essa regra é semelhante à que se aplica na definição das propriedades da classe. Então, nada de usar *underscore* para indicar a visibilidade do método.

Para exemplificar, vamos ao exemplo a seguir que demonstra a maneira correta de definir um método:

```
<?php
```

```php
namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    private function getNome()
    {
        return $this->nome;
    }

    protected function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

Provavelmente você deve estar se perguntando: "Qual o sentido de ter os métodos `setNome($nome)` e `getNome()` protegido e privado?". Na verdade não tem sentido mesmo, é apenas para mostrar que os modificadores de acesso `private` e `protected` existem, podem e devem ser utilizados na definição do método.

Agora veremos um exemplo considerado incorreto:

```php
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    function _getNome()
    {
        return $this->nome;
    }

    function _setNome($nome)
    {
        $this->nome = $nome;
```

```
    }
}
```

O exemplo funciona, mas não é correto, pois os modificadores de acesso existem para serem utilizados. Então use e abuse deles e evite utilizar `_` (*underscore*) como modificador de acesso pois seu código não ficará em conformidade com a PSR-2, ok?

Regra 22

Após o nome do método NÃO DEVE existir um espaço em branco. A chave de abertura do método DEVE estar em uma nova linha e a chave de fechamento DEVE ficar logo após o corpo do método.

Após o parêntese de abertura NÃO DEVE existir um espaço em branco e antes do parêntese de fechamento também NÃO DEVE haver um espaço em branco.

Pode parecer estranho, mas isso é considerado mais uma questão de adaptação e costume. Constantemente vejo códigos que não seguem essa regra da PSR. Para facilitar o entendimento, vamos ao exemplo considerado correto:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction()
    {
        //Código
    }
}
```

No exemplo citado, veja que o método `indexAction()` segue

perfeitamente essa regra da PSR-2: não há espaços após o nome do método, a chave de abertura está em uma nova linha e a chave de fechamento está logo após o conteúdo do método.

Vamos analisar agora um exemplo em que a definição do método não segue o padrão estipulado:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction ( $request ) {
        //Código
    }
}
```

Apesar de o código funcionar sem problemas, ele não é compatível com a regra da PSR-2.

Conclusão

Como você pôde ver são inúmeras regras que a PSR-2 disponibiliza, e ainda tem mais! No próximo capítulo continuaremos vendo quais são as demais regras da PSR-2. Então siga em frente!

Capítulo 4

PSR-2 (CODING STYLE GUIDE) - PARTE II

Vimos no capítulo anterior um total de 22 regras da PSR-2, neste capítulo vamos continuar vendo essas regras que visam facilitar o estilo do nosso código-fonte para obtermos uma melhor leitura e compreensão através dos padrões estabelecidos pela PSR-2.

4.1 REGRAS

A seguir vamos continuar com as regras de onde paramos no capítulo anterior.

Regra 23

Quando o método possuir uma lista de parâmetros, NÃO DEVE haver espaço antes de cada vírgula, mas DEVE ter espaço após cada vírgula. Bem simples, não é mesmo? Sempre que você for definir uma lista de parâmetros no método, você deve separá-los por vírgula (,) e em seguida, um espaço, seguindo basicamente o padrão já usado em nosso próprio idioma.

Vamos ao exemplo considerando a maneira correta de definir

os parâmetros do método:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction($request, $response)
    {
        //Código
    }
}
```

O exemplo mostra exatamente como a regra explica, cada parâmetro ou argumento é separado por vírgula seguida de um espaço (,). Agora vamos ver um exemplo considerado como a maneira incorreta na definição dos parâmetros de um método:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction($request ,$response)
    {
        //Código
    }
}
```

Perceba que dessa vez os parâmetros são separados apenas pela vírgula, porém o espaço que deveria estar após a vírgula está definido antes dela. Sendo assim, o código não está em conformidade com a PSR-2.

Regra 24

Caso o método possua muitos parâmetros a serem definidos, a

lista PODE ser dividida em múltiplas linhas contendo uma indentação, sendo que o primeiro parâmetro da lista DEVE estar na próxima linha.

DEVE haver apenas um parâmetro por linha e o parêntese de fechamento da lista e a chave de abertura do método DEVEM estar na mesma linha, contendo apenas um espaço entre eles. A definição dessa regra pode ser grande, mas não é difícil. Na verdade, na primeira vez em que comecei a usar a PSR-2 eu estranhei bastante a forma como ficava o código, mas com o tempo acabei me acostumando.

A explicação desta regra fica mais fácil ao visualizar um exemplo que atenda à especificação:

```
<?php

namespace App\Sercice\Paginacao;

class PaginacaoService
{
    public function paginate(
        $limite,
        $pagina,
        $dados
    ) {
        //Código
    }
}
```

Perceba como a lista de parâmetros foi dividida, e ficou fácil de ler e identificar os parâmetros. Agora vamos analisar o mesmo exemplo em que a divisão em múltiplas linhas não é correta e deve ser evitada:

```
<?php

namespace App\Sercice\Paginacao;
```

```
class PaginacaoService
{
    public function paginate(
        $limite, $pagina,
        $dados
    ) {
        //Código
    }
}
```

O código do exemplo funciona perfeitamente, porém não está em conformidade com a regra de se ter cada parâmetro em uma nova linha, então evite esse tipo de estilo.

Regra 25

As palavras-chaves `abstract` e `final` DEVEM ser declaradas antes de definir a visibilidade de classes e métodos. Há casos em que precisamos criar uma classe abstrata ou final, por exemplo, ou até mesmo métodos, e existe uma maneira correta de se fazer isso.

Para simplificar o entendimento, vamos analisar o exemplo a seguir que demonstra a maneira correta de seguir essa regra:

```
<?php

namespace App\Sercice;

abstract class AbstractService
{
    protected static $dados;

    abstract protected function insert();

    final public static function start()
    {
        //Código
```

```
    }
}
```

Perceba que as palavras-chaves `abstract` e `final` precedem a visibilidade dos dois métodos `insert()` e `start()`. Na classe, não existe a visibilidade para ser definida, mas podemos informar se é uma classe abstrata ou final.

É um exemplo simples, mas que demonstra a utilização dessas palavras-chaves. Agora veremos um exemplo que demonstra a maneira incorreta de utilizá-las:

```
<?php

namespace App\Sercice;

class abstract AbstractService
{
    protected static $dados;

    protected abstract function insert();

    public final static function start()
    {
        //Código
    }
}
```

Nesse exemplo, as palavras-chaves `abstract` e `final` encontram-se logo após a visibilidade dos métodos e também logo após a palavra-chave `class`.

Esse exemplo já não funcionará pois não podemos definir uma classe abstrata após a palavra-chave `class`, apenas antes.

Perceba que nos métodos também existem as palavras `abstract` e `final` após a definição da visibilidade, e não há problemas. Porém, se fizer deste modo, seu código não ficará de

acordo com a regra estabelecida.

Regra 26

Ao declarar um método ou função NÃO DEVE existir espaço entre o nome do método/função e o parêntese de abertura.

NÃO DEVE existir espaço após o parêntese de abertura e NÃO DEVE existir espaço antes do parêntese de fechamento; e na lista de parâmetros NÃO DEVE existir espaço antes de cada vírgula, mas DEVE existir um espaço após cada vírgula. Parece complexo, não é mesmo? Mas não é complicada - na verdade, é tão simples quanto definir uma variável. Veja o exemplo a seguir que demonstra a maneira correta da aplicação dessa regra:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        //Código
    }
}
```

Simples, não? Note que, quando definimos o nome do método, não existe espaço antes e após o parêntese de abertura, não existe espaço antes da vírgula entre os parâmetros e não existe espaço antes e após o parêntese de fechamento.

Agora vamos analisar um exemplo com a maneira incorreta e que não estará em conformidade com essa regra da PSR:

```
<?php
```

```php
namespace App\Sercice;

class AbstractService
{
    protected function update ( $id , array $dados )
    {
        //Código
    }
}
```

Esse exemplo está completamente fora da regra, pois existe espaço em tudo quanto é lugar na definição do método, evite ao máximo utilizar esse estilo de código.

Regra 27

Após a palavra-chave da estrutura de controle DEVE existir um espaço. NÃO DEVE existir um espaço após o parêntese de abertura (se necessário).

NÃO DEVE existir espaço antes do parêntese de fechamento (se necessário). DEVE existir um espaço entre o parêntese de fechamento e a chave de abertura.

O corpo da estrutura de controle DEVE ser indentado uma vez. A chave de fechamento DEVE ser colocada uma linha após o corpo da estrutura de controle.

Essa regra também parece assombrosa, mas ela é simples. Volto a dizer que é mais uma questão de adaptação e costume do que uma dificuldade lógica propriamente dita.

Vamos analisar um exemplo para cada estrutura de controle, que demonstra a maneira correta a ser seguida:

Exemplos com `if` , `elseif` , `else`

```php
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        if (is_int($id)) {
            $resultado = 'ID é um inteiro';
        } elseif (is_numeric($id)) {
            $resultado = 'ID é uma string contendo apenas números
;
        } else {
            $resultado = 'ID não possui um formato válido';
        }
    }
}
```

Perceba a indentação da estrutura de controle e também o espaço antes do parêntese de abertura e, ainda, o espaço após o parêntese de fechamento antes da chave de abertura da estrutura de controle.

Não é uma regra complicada a ser seguida, mas deve-se tomar cuidado para não definir incorretamente a estrutura de controle e é isso que veremos no exemplo a seguir, que demonstra a maneira considerada incorreta pela PSR e que deve ser evitada:

```php
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;
```

```
        if(is_int($id)){
            $resultado = 'ID é um inteiro';
        }
        elseif(is_numeric($id)){
            $resultado = 'ID é uma string contendo apenas números
;
        }
        else{
            $resultado = 'ID não possui um formato válido';
        }
    }
}
```

Já nesse exemplo, apesar de funcionar corretamente, ele não segue a regra da PSR. Como você pode ver, não existe espaço antes do parêntese de abertura e nem após o parêntese de fechamento.

Outra regra quebrada é quanto às palavras-chaves `elseif` e `else` que estão abaixo da chave de fechamento da estrutura anterior. O correto é estarem na mesma linha que a chave de fechamento e precedidas de um único espaço. Nada demais, não é mesmo?

Exemplos com `switch`, `case`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        switch ($id) {
            case 0:
                $resultado = 'Id é 0';
                break;
            case 1:
                $resultado = 'Id é 1';
```

```
                break;
            case 2:
                $resultado = 'Id é 2';
                break;
            case 3:
                $resultado = 'Id é 3';
                //Sem break
            default:
                $resultado = 'O valor informado não é permitido';
                break;
        }
    }
}
```

Perceba como é simples: após a palavra-chave `switch` DEVE existir um espaço antes do parêntese de abertura e DEVE existir um espaço após o parêntese de fechamento.

Cada `case` DEVE ser indentado uma única vez e o conteúdo de cada `case` também DEVE ser indentado uma única vez. A palavra-chave `break` DEVE estar indentada no mesmo nível do conteúdo do `case` .

Quando não houver `break` DEVE existir um comentário `//no break` no caso.

Este exemplo está perfeitamente em conformidade com a PSR e deve ser utilizado dessa maneira.

Agora vamos analisar um exemplo que é considerado incorreto quanto ao estilo aplicado na regra:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
```

```
    {
        $resultado = null;

        switch($id){
            case 0:
                $resultado = 'Id é 0';
            break;
            case 1:
                $resultado = 'Id é 1';
            break;
            case 2:
                $resultado = 'Id é 2';
            break;
            case 3:
                $resultado = 'Id é 3';
            default:
                $resultado = 'O valor informado não é permitido';
            break;
        }
    }
}
```

Perceba que nesse exemplo a estrutura sofreu alterações de indentação e espaçamento, e também que no `case` que não possui o `break` o comentário `//no break` foi removido.

O método continua funcionando perfeitamente, porém não é válido para a PSR-2.

Exemplos com `while`, `do while`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        while ($id < 10) {
            //Código
        }
```

```
        do {
            //Código
        } while ($id > 0)
    }
}
```

Após a palavra-chave `while` , DEVE existir um espaço antes do parêntese de abertura e DEVE existir um espaço após o parêntese de fechamento.

A abertura da chave deve estar na mesma linha da declaração `while` . O mesmo se aplica utilizando o `do while` .

O código fica mais bonito e legível, e essa é a maneira considerada correta pela PSR. Agora vamos ver o exemplo a seguir que demonstra a maneira incorreta e que não deve ser seguida:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        while($id < 10){
            //Código
        }

        do{
            //Código
        }while($id > 0)
    }
}
```

O exemplo é semelhante ao da maneira correta, a única diferença aqui são os espaçamentos que não existem. O exemplo funciona normalmente, mas não é aceito pela PSR.

Exemplos com `for`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        for ($i = 0; $i <= 10; $i++) {
            //Código
        }
    }
}
```

Nesse exemplo foi aplicado exatamente como no exemplo da utilização correta com `while`, ou seja, os espaçamentos estão de acordo com a regra da PSR.

Perceba novamente que, após a palavra-chave `for`, DEVE existir um espaço antes do parêntese de abertura e DEVE existir um espaço após o parêntese de fechamento.

Vamos analisar agora o exemplo considerado incorreto e que não é aceito pela PSR:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        for($i = 0; $i <= 10; $i++){
            //Código
        }
    }
}
```

Nesse exemplo, o código também funciona corretamente, porém não é aceito pela PSR pois não possui os espaçamentos definidos.

Exemplo com `foreach`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        foreach ($dados as $key => $value) {
            //Código
        }
    }
}
```

Esse exemplo é semelhante ao aplicado com o `for` e as mesmas regras de espaçamento também se aplicam utilizando o `foreach`. Esse exemplo possui os espaçamentos que são considerados corretos pela PSR e deve ser utilizado.

Vamos analisar agora o mesmo exemplo, porém com a ausência dos espaçamentos e que é considerado incorreto:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        foreach($dados as $key => $value){
            //Código
        }
    }
}
```

Nada demais até aqui, o exemplo funciona normalmente, mas não está de acordo com a PSR.

Exemplos com `try`, `catch`, `finally`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        try {
            //Código
        } catch (\InvalidArgumentException $e) {
            //Código
        } catch (\Exception $e) {
            //Código
        } finally {
            //Código
        }
    }
}
```

Quando houver a necessidade de utilizar o bloco `try catch` ou `try catch finally`, ele deverá ter a estrutura de espaçamentos e indentação mostrados no exemplo anterior.

Após a palavra-chave `try` DEVE existir um espaço antes da abertura da chave e DEVE existir um espaço após a chave de fechamento.

Para cada bloco `catch` DEVE existir um espaço após a palavra-chave `catch` e DEVE existir um espaço após o parêntese de fechamento.

O mesmo se aplica ao `finally` (quando existir): após a palavra-chave `finally`, DEVE existir um espaço antes da chave

de abertura do bloco.

Esse exemplo está completamente em conformidade com a regra da PSR. Vamos analisar um exemplo em que essa regra não é seguida e é considerado incorreto:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        try{
            //Código
        }
        catch(\InvalidArgumentException $e){
            //Código
        }
        catch(\Exception $e){
            //Código
        }
        finally{
            //Código
        }
    }
}
```

No exemplo anterior todo o código foi alterado e não há espaçamento onde deveria ter e a indentação também está incorreta, não sendo aceito pela PSR.

Regra 28

Closures (funções anônimas) DEVEM ser declaradas com um espaço após a palavra-chave `function` e também DEVE haver um espaço antes e depois da palavra-chave `use` .

A chave de abertura DEVE estar na mesma linha e a chave de fechamento DEVE estar na linha após o conteúdo da função.

Para simplificar o que a regra diz, vamos ao exemplo considerado correto:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function ($atualizado) use ($id) {
            //Código
        }
    }
}
```

Perceba como a estrutura de uma `closure` (função anônima) é definida. Esse código faz exatamente como a regra define: após a palavra-chave `function` existe um espaço antes do parêntese de abertura e outro após o parêntese de fechamento.

Após a palavra-chave `use` também existe um espaço antes do parêntese de abertura e outro após o parêntese de fechamento.

Também é importante notar que a chave de abertura da função está na mesma linha de sua declaração e a chave de fechamento está logo após seu conteúdo. Agora vamos analisar um exemplo que é considerado incorreto pela PSR e não deve ser seguido:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
```

```
    public function updateAction($id, array $dados)
    {
        return function($atualizado) use($id)
        {
            //Código
        }
    }
}
```

No exemplo, o código não está em conformidade com a regra e não deve ser utilizado pois a ausência de espaço e indentação não é correta.

Regra 29

Após o parêntese de abertura, NÃO DEVE existir espaço e NÃO DEVE existir um espaço antes do parêntese de fechamento.

NÃO DEVE existir espaço antes de cada vírgula na lista de parâmetros, mas DEVE existir um espaço após cada vírgula.

Caso existam parâmetros com valor padrão, ele DEVE ser definido no final da lista de parâmetros. Basicamente aplicamos essa regra no exemplo anterior considerado correto.

Vamos analisar a aplicação dessa regra no exemplo a seguir:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function ($atualizado, $usuario) use ($id, $dados)
{
            //Código
        }
```

```
    }
}
```

Como pode ser visto, após a palavra-chave `function` existe um espaço antes do parêntese de abertura, mas não existe espaço após.

Também não existe espaço antes da vírgula e também não existe espaço antes do parêntese de fechamento, tornando-o um exemplo a ser seguido. Vamos ver agora um exemplo considerado incorreto e que não deve ser utilizado:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function ( $atualizado , $usuario ) use ( $id , $d
ados ) {
            //Código
        }
    }
}
```

Não é necessário descrever mais nada, não é mesmo? Perceba que há espaços em tudo quanto é canto, tornando o exemplo inviável.

Regra 30

Assim como nos métodos, a lista de parâmetros da `closure` (função anônima) PODE ser dividida em múltiplas linhas, sendo que cada linha DEVE ser indentada uma única vez.

Sempre que isso ocorrer, o primeiro parâmetro DEVE estar na

próxima linha e DEVE existir apenas um parâmetro por linha.

Quando a lista de parâmetros for dividida em múltiplas linhas, o parêntese de fechamento e a chave de abertura DEVEM estar na mesma linha, contendo apenas um espaço entre ambos.

Difícil? Não se preocupe, já vimos o funcionamento quando aplicamos essa regra utilizando métodos. Vamos ao exemplo considerado correto pela PSR e que deve ser seguido para que seu código esteja em conformidade:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function (
            $atualizado,
            $usuario
        ) use (
            $id,
            $dados
        ) {
            //Código
        }
    }
}
```

Veja como o código é dividido e indentado seguindo corretamente a regra estabelecida pela PSR. Vamos analisar um exemplo considerado incorreto:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
```

```
{
    public function updateAction($id, array $dados)
    {
        return function ($atualizado,
            $usuario) use ($id,
            $dados) {
            //Código
        }
    }
}
```

O exemplo anterior demonstra exatamente o que não fazer. Não indente o código dessa maneira, por mais que funcione. Se você fizer isso, seu código não estará em conformidade com a PSR.

Conclusão

Chegamos ao final da PSR-2 e, como pôde ser visto neste capítulo, há várias regras e condições que devem ser seguidas para que seu código esteja em conformidade. Aplicamos regras de espaços e indentação que mostraram que o código fica bem mais fácil de ser entendido.

No primeiro momento pode parecer estranho como o código fica, mas com o tempo você acaba se acostumando e até mesmo decorando como cada regra é aplicada.

É importante ressaltar que a PSR-2 foi descontinuada e a sua substituta é a PSR-12, que veremos mais adiante neste livro.

Vamos seguir em frente e conhecer a PSR-3 e suas regras.

Capítulo 5

PSR-3 (LOGGER INTERFACE)

Com o objetivo de fornecer uma interface comum para gravar logs de maneira simples, fácil e universal, a PSR-3 permite que bibliotecas recebam um objeto `Psr\Log\LoggerInterface`. Com ele, é possível criar logs personalizados para cada aplicação.

Aplicações que possuem necessidades específicas PODEM estender a interface, mas DEVEM permanecer compatíveis. Ou seja, cada aplicação poderá ter uma maneira diferente de gravar log, mas independentemente da aplicação a compatibilidade deve ser mantida.

Assim como outras PSRs, essa também possui uma série de regras que devem ser seguidas e é exatamente elas que veremos na seção a seguir:

5.1 REGRAS

Confira nesta seção o conjunto de todas as regras e características dessa PSR.

Regra 1

A interface `LoggerInterface` possui oito métodos para gravação dos logs para os oito níveis. São eles: `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert`, `emergency`.

Um nono método chamado `log` também é definido na interface. O método `log` possui exatamente o mesmo objetivo que os demais níveis. A diferença é que ele aceita como primeiro parâmetro um dos oito níveis definidos, ou seja, o resultado DEVE ser exatamente o mesmo que ao chamar o método do nível específico.

Veja o exemplo a seguir, que demonstra o funcionamento dessa regra:

```
log('error', 'Erro grave ocorreu', array $context = array());
```

No exemplo anterior, mostramos o funcionamento do método `log` que possui exatamente o mesmo funcionamento do que ao chamarmos o método `error` diretamente. Isso é possível porque estamos passando para ele em qual nível queremos gravar o `log`.

Com essa flexibilidade, você pode escolher a qual dos dois modos você vai aderir em sua aplicação. Isto é, se prefere utilizar diretamente o método de algum nível definido na `interface`, ou o método `log`, pois o comportamento será o mesmo.

Se o nível passado como parâmetro não for válido, uma exceção `Psr\Log\InvalidArgumentException` deverá ser lançada.

Regra 2

Todos os métodos de log DEVEM aceitar uma string ou um objeto com o método `__toString()` como mensagem de log.

Métodos PODEM ter tratamentos específicos para os objetos recebidos ou, se não for o caso, então os métodos DEVEM transformar o objeto em string.

Na verdade é bem simples: quando você for passar a mensagem do log, além de poder passar uma string, você poderá passar um objeto que contenha o método `__toString()`, garantindo que o objeto seja convertido em string.

Regra 3

A mensagem PODE conter espaços reservados que PODEM ser substituídos por valores contidos no parâmetro `$context`.

Os nomes dos marcadores DEVEM ser iguais às chaves contidas no parâmetro `$context`. Os nomes dos marcadores DEVEM ser definidos com uma chave de abertura (`{`) e uma chave de fechamento (`}`).

NÃO DEVE existir espaço em branco entre os delimitadores e o nome do marcador. Os nomes dos marcadores DEVEM ser compostos apenas pelos caracteres A-Z, a-z, 0-9, underscore (_) e ponto.

Dependendo da aplicação, pode haver a necessidade de ter um log mais personalizado, contendo informações de usuários como nome, data de acesso etc.

Para esses casos, podemos utilizar os marcadores, definindo uma mensagem e passando no contexto o valor desse marcador. Para facilitar o entendimento, vamos analisar o exemplo a seguir, que demonstra a aplicação dessa regra:

```php
<?php

namespace App\Service\Log;

use Psr\Log\LoggerInterface;

class LogService implements LoggerInterface
{
    //Código implementado

    public function gravarLog()
    {
        $mensagem = 'O usuário {NOME_USUARIO} acessou o sistema';
        $contexto = [
            'NOME_USUARIO' => 'Jhones'
        ];

        $this->info($mensagem, $contexto);
    }
}
```

Perceba que o nome do marcador no exemplo é `NOME_USUARIO` e é exatamente esse nome que vou colocar como chave dentro do array de contexto. Ou seja, a chave dentro do array de contexto se chama `NOME_USUARIO` definida com o valor `Jhones`. O marcador `NOME_USUARIO` será substituído pelo nome `Jhones`.

Regra 4

Métodos PODEM implementar diversas estratégias para escapar valores de marcadores e tradução de logs para exibição.

Os usuários NÃO DEVEM pré-escapar os valores dos marcadores caso não conheçam em quais contextos os dados serão exibidos. Você não deve substituir o marcador por algum texto ou informação sem realmente conhecer a fundo que a aplicação não irá utiliza-lo em algum outro local. Você já deve ter visto por aí que há muitos escapes (substituição) de valores de variáveis, e em um sistema de log não é diferente.

É permitido fazer esse escape, mas no momento de gravar o log o valor não deve ser alterado, mas deve ser enviado da maneira que foi gerado sem escapes, por exemplo.

Regra 5

Todos os métodos aceitam um array de contexto como parâmetro, sendo possível passar qualquer tipo de informação adicional sobre o log.

Os métodos DEVEM tratar esse array da maneira que for mais conveniente para a aplicação, sendo que, seja qual for o valor passado no array de contexto, estes NÃO DEVEM causar erros ou lançar uma exceção.

Aqui não tem nada demais, caso o parâmetro de contexto possua algum valor que não tenha sido utilizado ou tratado, esse valor poderá ser ignorado ou armazenado para uma utilização futura, de modo que não ocasione erros.

Regra 6

Se dentro do array de contexto for passado um objeto do tipo `Exception` , ele DEVERÁ estar armazenado dentro da chave

`exception` do array.

Logs de exceções são bastante úteis pois nos ajudam a acompanhar a pilha da exceção, tornando o trabalho de correção do erro mais ágil.

Quando existir a chave `exception` dentro do array de contexto, os métodos DEVEM verificar se realmente é uma exceção antes de fazer a utilização do valor, já que a chave `exception` PODE conter qualquer outro tipo de valor que não seja uma exceção.

Se for necessário gravar logs de exceção, você deverá passar o objeto `Exception` dentro da chave `exception` do array de contexto.

Atente-se também ao fato de que, se um determinado método precisar acessar essa exceção, o método tem que ter um tratamento adequado para garantir que o valor dentro da chave `exception` é realmente um objeto `Exception` .

5.2 CLASSES E INTERFACES

Agora vamos conhecer os integrantes desse pacote. São eles:

- `Psr\Log\AbstractLogger` - Essa classe abstrata possui a implementação da interface `LoggerInterface` , que possui os nove métodos, oito dos quais são referentes aos níveis de log permitidos por padrão, para que a aplicação possa estender a classe abstrata que já contém todos os métodos necessários para serem utilizados.

- `Psr\Log\LoggerTrait` - Funciona basicamente da mesma maneira que a classe `AbstractLogger`, tendo como diferença o fato de que a `Trait` não implementa a interface `LoggerInterface` porque as `Traits` não podem implementar interfaces. Por conter o mesmo funcionamento que a classe abstrata, o resultado será exatamente o mesmo, já que a `LoggerTrait` já possui o método abstrato `log`.

- `Psr\Log\NullLogger` - Estende a classe abstrata `AbstractLogger` e é responsável por ser uma espécie de contenção caso nenhuma classe responsável pela gravação de log na aplicação tenha sido definida.

- `Psr\Log\LoggerAwareInterface` - Essa interface possui apenas um método chamado `setLogger`, que é responsável por setar uma determinada classe de logs.

- `Psr\Log\LoggerAwareTrait` - Funciona como uma implementação da interface `LoggerAwareInterface`, que provê o método `setLogger` e também a propriedade `$this->logger`.

- `Psr\Log\LogLevel` - Simplesmente contém as oito constantes referentes aos níveis de log.

Conclusão

Chegamos ao final de mais uma PSR e vimos através dela como podemos desenvolver um sistema de logs seguindo as regras descritas aqui. Essa PSR não é tão grande e também não é tão complicada após conhecer as regras que a define.

É importante ressaltar que a PSR-3 não descreve um passo a passo de como desenvolver um sistema de logs, mas sim, nos dá o direcionamento e a estrutura inicial de como podemos implementar em nossa aplicação.

Capítulo 6

PSR-4 (AUTOLOADER)

Com o objetivo de fornecer um padrão para o carregamento de classes através de caminhos de arquivos, a PSR-4 é totalmente interoperável, podendo ser utilizada em conjunto com qualquer outro tipo de PSR. Além disso, a PSR-4 descreve onde podemos colocar os arquivos que serão carregados automaticamente.

6.1 REGRAS

Confira nesta seção o conjunto de todas as regras e características dessa PSR.

Regra 1

O nome completo da classe DEVE possuir um namespace de nível superior, conhecido como *namespace raiz*. Caso você não conheça ou não saiba como funciona, um namespace raiz nada mais é do que um namespace base em que todas as nossas classes (incluindo *traits* e interfaces) devem ser incluídas. Vamos analisar o exemplo a seguir para melhor entendimento:

```
<?php

namespace App\Validator\Cpf;
```

```
class CpfValidator
{
    //Código
}
```

No exemplo, foi criada uma classe `CpfValidator`, que por sua vez está dentro do namespace raiz `App`, onde `App` é o nome dado ao projeto ou ao módulo dele.

Se fôssemos criar outra classe, ela também deveria ficar dentro do namespace raiz `App`. Se você nunca ouviu falar ou nunca trabalhou com namespaces, deve estar se perguntando: mas posso definir qualquer nome para o namespace raiz? A resposta é SIM!

Se você quiser chamar o seu namespace raiz de `Sistema` ou até mesmo colocar seu nome, você pode fazer isso sem problemas, mas lembre-se de que o namespace deve fazer sentido prático. Geralmente é o nome da aplicação ou o nome do módulo. No nosso exemplo, isso significa que a estrutura de diretórios seria:

```
|-src
    |-App
        |-Validator
            |-Cpf
                |-CpfValidator.php
```

Simples, não é?

Regra 2

O nome completo da classe PODE ter um ou mais *subnamespaces*. Entenda como um subnamespace como sendo um subdiretório em uma estrutura de diretórios. No exemplo anterior você pôde ver que dentro do namespace raiz `App` temos ainda o `Validator` e o `Cpf`, então o namespace completo fica da seguinte maneira: `App\Validator\Cpf`.

Regra 3

O nome completo da classe DEVE terminar com o nome de uma classe. Ainda, seguindo o último exemplo citado, `App\Validator\Cpf`, quando formos chamar esse `validator` dentro de uma outra classe, faremos da seguinte maneira:

```
<?php

namespace App\Service\Usuario;

use App\Validator\Cpf\CpfValidator;

class UsuarioService
{
    //Código
}
```

O trecho `use App\Validator\Cpf\CpfValidator` está dizendo que estamos importando a classe `CpfValidator`. Ou seja, para o *autoloader* chamar uma classe, ele sempre vai considerar o que estiver após a última `\`, sendo o nome da classe somado ao namespace completo.

Regra 4

Underscores (_) NÃO DEVEM ter efeito especial no nome da classe. Significa que o underscore no nome de uma classe (incluindo *traits* e interfaces) deve ser simplesmente um caractere, sem causar nenhum tipo de efeito especial.

Regra 5

O nome completo da classe PODE conter qualquer tipo de combinação de caracteres em maiúsculos ou minúsculos. Isso

significa que não existe uma regra para definição do nome da classe incluindo o namespace, você pode utilizar qualquer tipo de combinação de letras.

Por convenção, o padrão adotado para nomes de classes e namespaces é o `StudlyCaps`, ou seja, a primeira letra de cada palavra é em maiúsculo, por exemplo: `App\Entity\Acl\PapelDeAcesso`.

Regra 6

Todos os nomes de classes DEVEM ser referenciados de maneira sensível (*case-sensitive*). Sempre que o *autoloader* chamar uma classe, ele utilizará o nome real da classe, respeitando a diferenciação das letras. Por exemplo, `CpfValidator` é diferente de `cpfValidator`, sendo que neste último caso a classe não seria encontrada, pois o nome real dela é `CpfValidator` e não `cpfValidator`.

Regra 7

Ao carregar um nome de arquivo correspondente a um nome completo de classe, uma série de um ou mais namespaces com exceção do namespace principal DEVE corresponder a um diretório raiz. Já vimos isso em um dos exemplos anteriores: todo namespace deve ser correspondente a um diretório raiz com exceção do namespace raiz, afinal ele já é a raiz.

Mas vamos analisar novamente a estrutura de diretórios de um dos exemplos anteriores:

```
|-src
    |-App
```

```
|-Validator
    |-Cpf
        |-CpfValidator.php
```

Perceba que o diretório `App` é a base de todos os subdiretórios subsequentes, e a estrutura de diretórios é convertida em formato de namespace `App\Validator\Cpf\CpfValidator`.

Regra 8

Cada subnamespace corresponde a um subdiretório incluso no diretório raiz e o nome do subdiretório DEVE ser escrito da mesma maneira que o nome do subnamespace, sempre respeitando a diferenciação de letras.

Não há segredo: o nome do namespace deverá será ser definido exatamente da maneira que for definido o nome do diretório, conforme mostrado anteriormente.

Regra 9

A classe que for chamada após o namespace DEVE corresponder a um arquivo `.php` e o nome desse arquivo DEVE ser escrito da mesma maneira que o nome da classe, sempre respeitando a diferenciação de letras.

Também já falamos disso anteriormente. No caso da nossa classe `CpfValidator`, note que ela segue exatamente o que descreve essa regra. Ela está sendo chamada por um namespace base e ainda respeita exatamente o mesmo nome do arquivo contido no diretório, diferenciando as letras maiúsculas e minúsculas.

Regra 10

Implementações de *autoloader* NÃO DEVEM lançar exceções, NÃO DEVEM gerar erros de qualquer nível e NÃO DEVERIAM retornar nenhum valor. O único objetivo do *autoloader* é fazer o carregamento automático, nada além disso.

Para evitar erros, o desenvolvedor deve saber exatamente os namespaces que deverá utilizar de modo que não ocasione nenhum tipo de exceção ou erro.

Conclusão

Podemos concluir que a PSR-4 é uma especificação que define o carregamento automático de classes através de arquivos contidos em diretórios e subdiretórios que formam um determinado namespace.

Cada namespace deve seguir exatamente a mesma estrutura de diretórios e a mesma diferenciação de letras maiúsculas e minúsculas.

Nosso próximo passo é falar sobre a PSR-6, e por isso, talvez você possa estar se perguntando, mas o que aconteceu com a PSR-5? Até o momento da escrita deste livro a PSR-5 (PHPDoc Standard) está em faze de esboço e por isso não será tratada aqui. Focamos apenas nas PSRs que foram aprovadas e estão ativas no momento, tudo bem? Excelente, então vamos nessa rumo à PSR-6.

Capítulo 7

PSR-6 (CACHING INTERFACE)

Fazer cache dos dados da aplicação se tornou cada vez mais necessário, porém há muitas maneiras de se fazer isso. Cada framework e/ou microframework possui uma forma diferente de fazer esse gerenciamento e uma das principais consequências disso é que o desenvolvedor tem que ficar se adaptando a cada novo modelo de definir o cache. No Zend Framework é de um jeito, no Symfony é de outro e por aí vai.

Com o objetivo de acabar com esse problema, surgiu a PSR-6 que visa definir um padrão para criação de cache e é exatamente isso que veremos a seguir.

7.1 REGRAS

Confira nesta seção o conjunto de todas as regras e características que fazem parte dessa PSR.

Regra 1

Chamando na aplicação - A aplicação ou o código que utilizar os serviços de cache DEVERÁ implementar a interface padrão,

caso contrário, a aplicação não saberá da existência do serviço de cache.

Isso significa que, se a sua aplicação for implementar o cacheamento de dados seguindo esta PSR, então deverá ter uma classe que realizará a implementação da interface padrão de cache definida, que no caso é a interface `Psr\Cache\CacheItemPoolInterface`.

Regra 2

Implementando na aplicação - A aplicação é responsável por implementar o padrão estabelecido pela PSR com o objetivo de fornecer os serviços de cache para qualquer aplicação ou código de forma simples.

A implementação DEVE prover classes que implementam as interfaces `Psr\Cache\CacheItemPoolInterface` e `Psr\Cache\CacheItemInterface`. A implementação das bibliotecas DEVE suportar ao menos a funcionalidade TTL (*Time To Live*).

Isso significa que em uma biblioteca/aplicação ao menos a funcionalidade TTL deve ser suportada. Ela é basicamente obrigatória, já que define o tempo de vida de um item de cache.

Regra 3

TTL (Time To Live) - Cada item de cache possui o que chamamos de tempo de vida. Esse tempo de vida é a quantidade de tempo entre o momento em que esse item é armazenado e quando é considerado obsoleto.

O TTL é definido por um número inteiro que representa o tempo em segundos ou por um objeto `DateInterval` .

Regra 4

Expiração - É o tempo definido para que o cache se torne obsoleto, geralmente é calculado adicionando o TTL (*Time To Live*) ao tempo em que um determinado objeto é armazenado.

O tempo também pode ser adicionado através de um objeto `DateTime` . Por exemplo, um item de cache com um TTL de 300 segundos que foi armazenado às 09:00:00 terá um tempo de expiração em 09:05:00, ou seja, terá um tempo de 5 minutos antes de os dados em cache tornarem-se obsoleto.

A aplicação PODE expirar um item armazenado em cache antes do tempo de expiração definido, porém DEVE tratar um determinado item como expirado apenas quando o seu tempo de expiração for atingido.

Se a aplicação passar um item para ser armazenado e não especificar um tempo de expiração ou TTL, ela PODE utilizar um tempo de expiração padrão.

Caso nenhum tempo de duração for definido, então a aplicação DEVE interpretar como uma solicitação para armazenar o item em cache para sempre ou pelo tempo em que a implementação suportar.

Regra 5

Chave - É a chave que define um item de cache. Esse nome deve ser exclusivo para cada item de cache. As aplicações DEVEM

suportar as chaves que consistem nos caracteres de A-Z, a-z, 0-9, undescore (`_`) e ponto (`.`), e em qualquer ordem na codificação `UTF-8` , sendo que o nome da chave deve ter um comprimento de no máximo 64 caracteres.

A aplicação que implementar PODE suportar outros caracteres, outras codificações e/ou um nome de chave com um comprimento maior.

A aplicação também será responsável por escapar a sequência de caracteres conforme a necessidade, mas DEVE retornar a sequência original da chave sem nenhum tipo de modificação.

Os caracteres `{` , `}` , `(` , `)` , `/` , `\` , `@` e `:` NÃO DEVEM ser suportados pela aplicação, porque são reservados para extensões futuras.

Regra 6

Ocorrência - Uma ocorrência de cache é quando a aplicação solicita um item através da chave e um valor correspondente a essa chave é encontrado, sendo que esse valor não expirou e não é inválido.

Antes de utilizar a chave de um item de cache, a aplicação DEVE verificar se a consulta ao item de cache resultou em um `hit` (acerto). Esse procedimento é realizado através da utilização do método `isHit()` em todas as chamadas `get()` .

Regra 7

Falha/Erro - Ocorre quando a aplicação solicita um item de cache pela chave e o valor correspondente não é encontrado, ou o

valor é encontrado mas já expirou, ou até mesmo o valor é inválido por outros motivos. Sempre que um valor for considerado expirado, DEVERÁ ser considerado como um erro de cache.

Regra 8

Adiado - Um item de cache adiado indica que ele não pode ser armazenado imediatamente pelo pool. Um pool de objetos PODE adiar a persistência de um determinado item de cache para que seja possível aproveitar as operações em massa suportadas por alguns tipos de mecanismos de persistência.

Um pool DEVE garantir que quaisquer itens de cache adiados possam ser eventualmente persistidos e que os dados não sejam perdidos. Os pools de objetos PODEM persistir os itens de cache antes de a aplicação solicitar a gravação deles.

Quando a aplicação chamar o método `commit()` , todos os itens adiados DEVEM ser mantidos, a aplicação PODE utilizar a lógica mais adequada e apropriada para determinar quando persistir os itens de cache.

As solicitações de um item de cache que foram adiadas DEVEM retornar o item adiado mas que ainda não foi persistido.

Regra 9

A aplicação DEVE suportar todos os tipos de dados PHP que possam ser serializados. Isso inclui tipos como:

- **Strings** - Cadeia de caracteres de qualquer tamanho e em qualquer codificação compatível com o PHP.
- **Integers** - Todos os inteiros de qualquer tamanho

suportados pelo PHP, até 64 bits.

- **Floats** - Todos os pontos flutuantes.
- **Boolean** - `True` (Verdadeiro) e `False` (Falso).
- **Null** - O valor nulo real.
- **Arrays** - Arrays indexados, associativos e multidimensionais de qualquer profundidade.
- **Object** - Qualquer objeto que tenha suporte a serialização e desserialização sem perdas, tal que `$object == unserialize(serialize($object))`.

Objetos PODEM aproveitar os métodos mágicos `sleep()` ou `wakeup()` do PHP.

Regra 10

Todos os dados passados para a aplicação DEVEM ser retornados da mesma maneira como foram passados. Isso significa que, se a aplicação passar uma variável com o valor (string) "10", ela DEVERÁ retornar o valor (string) "10", e não um valor (int) 10.

A aplicação PODE utilizar as funções `serialize()` e `unserialize()` do PHP, mas não é obrigatório utilizá-las. A compatibilidade é simplesmente usada como uma base para objetos que são aceitos.

Se por algum motivo o valor exato não puder ser retornado, a aplicação DEVE retornar um erro de cache e não um erro de dados corrompidos/inválidos.

Regra 11

Pool - Representa uma coleção de itens em um sistema de

cache, ou seja, basicamente é um repositório que contém todos os itens de cache, e todos eles podem ser recuperados do pool como um objeto. Toda interação com os objetos que estiverem armazenados em cache acontecerá através do pool de objetos.

Regra 12

Itens - Um item é representado pelo par chave/valor dentro de uma coleção de dados, sendo que a chave é o principal identificador para um item e ela DEVE ser imutável, mas o valor pode ser alterado a qualquer momento ou sempre que necessário.

Regra 13

O controle de dados em cache geralmente é essencial e visa aumentar o desempenho da aplicação, mas nem por isso deve ser uma parte crítica que possa afetar as demais funcionalidades. Sendo assim um erro no cache NÃO DEVERIA resultar em falhas na aplicação.

Para o tratamento adequado dos erros provenientes de falhas de cache, a aplicação deve ser capaz de capturar todos os erros que possam ocorrer, mas NÃO DEVE lançar exceções que não foram definidas pela interface.

Ela DEVE ser capaz de capturar qualquer erro ou exceções que foram ocasionadas por outro sistema de armazenamento e NÃO DEVE permitir que esses dados se misturem.

A aplicação DEVE registrar esses erros e fazer o tratamento mais apropriado. Se for desejado que um ou mais itens do cache sejam excluídos, caso a chave especificada não exista, isso NÃO

DEVE ser considerado erro.

7.2 INTERFACES

A seguir vamos conhecer um pouco das interfaces que fazem parte dessa PSR:

`CacheItemInterface`

O objetivo dessa interface é definir um item de cache dentro da aplicação. Cada item de objeto DEVE estar associado a uma chave específica, que pode ser definida conforme as necessidades da aplicação.

A interface `Psr\Cache\CacheItemInterface` realiza o encapsulamento do armazenamento e a recuperação dos dados em cache, sendo que cada item do tipo `Cache\CacheItemInterface` armazenado em cache é gerado através de um objeto do tipo `Psr\Cache\CacheItemPoolInterface` , responsável por qualquer configuração necessária e também por associar o objeto a uma chave exclusiva.

Os objetos `Psr\Cache\CacheItemInterface` DEVEM ser capazes de armazenar e recuperar qualquer tipo de valor, conforme vimos anteriormente.

A aplicação NÃO DEVE realizar a chamada de itens diretamente, eles só podem ser chamados a partir de um pool de objetos, através do método `getItem()` .

A aplicação NÃO DEVERIA assumir que um item criado por ela mesma é compatível com um pool de objetos de outra biblioteca.

Vamos ver a seguir o modelo dessa interface disponibilizado pelo site oficial da PSR:

```
<?php

namespace Psr\Cache;

interface CacheItemInterface
{
    public function getKey();
    public function get();
    public function isHit();
    public function set($value);
    public function expiresAt($expiration);
    public function expiresAfter($time);
}
```

`CacheItemPoolInterface`

Possui o objetivo de aceitar uma chave passada pela aplicação e retornar o objeto `Psr\Cache\CacheItemInterface` associado à chave informada.

Também é o principal ponto de interação com toda a coleção de cache. É responsabilidade da aplicação realizar toda a configuração e inicialização do pool.

Vamos ver a seguir o modelo dessa interface:

```
<?php

namespace Psr\Cache;

interface CacheItemPoolInterface
{
    public function getItem($key);
    public function getItems(array $keys = array());
    public function hasItem($key);
    public function clear();
    public function deleteItem($key);
    public function deleteItems(array $keys);
```

```
    public function save(CacheItemInterface $item);
    public function saveDeferred(CacheItemInterface $item);
    public function commit();
}
```

`CacheException`

Essa interface se destina aos erros críticos que possam ocorrer, como configuração de cache, conexão com o servidor de cache, erros de credenciais etc.

Qualquer exceção lançada pela aplicação DEVE implementar essa interface.

```
<?php

namespace Psr\Cache;

interface CacheException
{
}
```

`InvalidArgumentException`

Essa interface estende a interface `Psr\Cache\CacheException` e é responsável por lançar uma exceção quando algum parâmetro/argumento inválido for passado para algum método.

```
<?php

namespace Psr\Cache;

interface InvalidArgumentException extends CacheException
{
}
```

Conclusão

Essa PSR realmente nos concede um bom direcionamento para quando formos desenvolver um sistema de cache para nossas aplicações.

Todas as regras dessa PSR que vimos até aqui nos mostram como nosso sistema de cache deve se comportar, seja quanto ao TTL, quanto à definição do par chave/valor, ou quando um erro ocorrer e por aí vai.

Seguindo tudo que foi descrito nela, não teremos problema e estaremos em conformidade com mais essa PSR.

No próximo capítulo falaremos sobre a PSR-7, que estipula um padrão para troca de mensagens HTTP, e essa é uma PSR muito utilizada principalmente por APIs, então vamos conhecê-la.

Capítulo 8

PSR-7 (HTTP MESSAGE INTERFACES)

O principal intuito dessa PSR é fornecer interfaces em comum para a troca de mensagens HTTP. Para que as aplicações possam se comunicar de maneira mais fácil, criou-se um padrão que é exatamente esse conjunto de interfaces definidas pela PSR-7 juntamente com suas regras.

Hoje muitos sistemas ainda se comunicam de maneiras diferentes e constantemente os desenvolvedores devem ficar estudando como uma determinada aplicação se comunica. Esta PSR visa tornar a nossa vida mais fácil também.

8.1 REGRAS

Confira nesta seção o conjunto de todas as regras e características que fazem parte dessa PSR.

Regra 1

Mensagens - Antes de prosseguirmos, você sabe o que é uma mensagem HTTP? Caso não saiba ou tenha dúvidas, aqui vai uma breve explicação.

Uma mensagem HTTP nada mais é do que uma requisição que uma aplicação faz para um servidor ou uma resposta de um servidor para a aplicação.

Essa PSR define interfaces para as mensagens HTTP `Psr\Http\Message\RequestInterface` e `Psr\Http\Message\ResponseInterface` respectivamente. Ambas as interfaces estendem de `Psr\Http\Message\MessageInterface`. Enquanto a interface `Psr\Http\Message\MessageInterface` PODE ser implementada diretamente, as classes concretas DEVEM implementar `Psr\Http\Message\RequestInterface` e `Psr\Http\Message\ResponseInterface`.

Regra 2

Cabeçalhos HTTP - O nome dos cabeçalhos deve ser `case-insensitive`, isto é, não deve fazer a distinção de letras maiúsculas e minúsculas. Por exemplo, o cabeçalho `Authorization` deve ser interpretado da mesma maneira que se fosse escrito `authorization`. Os cabeçalhos são recuperados através das classes que implementam a interface `Psr\Http\Message\MessageInterface`. Vamos analisar o exemplo a seguir, que demonstra exatamente o funcionamento desta regra:

```
<?php

namespace App\Middleware\Authorization

use Psr\Http\Message\MessageInterface;

class AuthorizationMiddleware
{
```

```
    private $mensagem;

    public function __construct(MessageInterface $mensagem)
    {
        $this->mensagem = $mensagem;
    }

    public function defineHeader()
    {
        $mensagem = $this->mensagem->withHeader('authorization',
'Bearer AbcDef1234@#');

        echo $mensagem->getHeaderLine('authorization');
        // Vai retornar: AbcDef1234@#

        echo $mensagem->getHeaderLine('AUTHORIZATION');
        // Vai retornar: AbcDef1234@#

        $mensagem = $this->mensagem->withHeader('Authorization',
'1234567890abcdef!@$');
        echo $mensagem->getHeaderLine('authorization');
        // Vai retornar: 1234567890abcdef!@$
    }
}
```

Como podemos ver no exemplo anterior, apesar de os cabeçalhos serem recuperados independentemente do tipo da escrita, sem diferenciar letras maiúsculas e minúsculas, o nome original DEVE ser mantido pela aplicação quando for recuperado por meio do método `getHeaders()`.

Os cabeçalhos também podem possuir múltiplos valores que podem ser definidos e recuperados de maneira simples e sem dor de cabeça.

Isso se dá por meio de uma instância de `Psr\Http\Message\MessageInterface`, que nos disponibiliza dois métodos: `getHeaderLine()` e `getHeader()`, sendo que o método `getHeaderLine()` retorna os valores contidos dentro do

cabeçalho em formato de `string`, enquanto o método `getHeader()` retorna os valores contidos no cabeçalho em forma de `array`.

Para simplificar o entendimento, vamos analisar o exemplo a seguir:

```
<?php

namespace App\Middleware\Authorization

use Psr\Http\Message\MessageInterface;

class AuthorizationMiddleware
{
    private $mensagem;

    public function __construct(MessageInterface $mensagem)
    {
        $this->mensagem = $mensagem;
    }

    public function defineHeaderValues()
    {
        $mensagem = $this->mensagem->withHeader('authorizatio
n', 'Bearer AbcDef1234@#')
            ->withAddedHeader('authorization', 'Bearer 123456
7890')

        $cabecalhoString = $mensagem->getHeaderLine('authoria
tion');
        //Retorna 'Bearer AbcDef1234@#, Bearer 1234567890'

        $cabecalho = $mensagem->getHeader('authorization');
        //Retorna ['Bearer AbcDef1234@#', 'Bearer 1234567890'

    }
}
```

Conforme podemos ver, definimos dois valores para um mesmo cabeçalho e os recuperamos de duas formas diferentes, em

forma de `array` e em forma de `string`.

Uma observação muito importante a ser considerada é que nem todos os valores podem ser concatenados usando uma vírgula (por exemplo, `Set-Cookie`).

Quando for trabalhar com esses cabeçalhos, a aplicação DEVE confiar no método `getHeader()` que está contido na interface `Psr\Http\Message\MessageInterface` para obter esses cabeçalhos com múltiplos valores.

Em requisições, o cabeçalho do host espelha o componente de host do URI (*Uniform Resource Identifier* ou Identificador de Recursos Universal, em português), assim como o host utilizado ao estabelecer a conexão TCP (*Transmission Control Protocol* ou Protocolo de Controle de Transmissão).

No entanto, a especificação do HTTP permite que o cabeçalho de host seja diferente de cada um dos dois. As aplicações DEVEM tentar definir o cabeçalho de host de um URI se nenhum cabeçalho dele for definido.

Por padrão, o método `withUri()` definido na interface `Psr\Http\Message\RequestInterface` substituirá o pedido retornado com um cabeçalho de host correspondente ao componente do host passado.

Você poderá optar por preservar o estado original do cabeçalho do host passando `true` para o segundo parâmetro do método `withUri()`.

Quando esse parâmetro é definido como `true`, o pedido retornado não atualizará o cabeçalho do host retornado pela

mensagem, a menos que ela não possua um cabeçalho de host definido.

Regra 3

Streams (Fluxos) - As mensagens HTTP consistem em uma linha inicial (método), cabeçalhos e um corpo, sendo que o corpo de uma mensagem HTTP pode possuir tamanhos variados, desde muito pequeno a extremamente grande.

Ao tentar representar o corpo de uma mensagem como uma `string` , por exemplo, a aplicação facilmente consome mais memória do que o necessário. Isso ocorre porque o corpo dessa mensagem deve ser armazenado completamente na memória.

Se a aplicação tentasse persistir o corpo de uma resposta ou requisição muito grande na memória, isso impossibilitaria a utilização dessa solução, pois facilmente seria apresentado um erro de memória (*Overflow*), ou seja, afetaria a performance da aplicação.

A interface `Psr\Http\Message\StreamInterface` é utilizada para ocultar os detalhes da implementação quando um determinado fluxo de dados é lido ou escrito. Além disso, essa interface expõe diversos métodos que permitem que os fluxos sejam lidos, escritos e percorridos de forma eficaz.

Os `streams` (fluxos) expõem seus recursos utilizando 3 métodos: `isReadable()` , que retorna se o fluxo é ou não legível, `isWritable()` , que retorna se o fluxo é ou não gravável e `isSeekable()` , que retorna se o fluxo é ou não procurado. Cada um pode ser utilizado para determinar se um fluxo de dados é

capaz de atender a seus requisitos ou não.

Cada instância do `stream` (fluxo de dados) terá diversos recursos, como somente para leitura, somente para gravação ou para leitura/gravação.

A instância do `stream` (fluxo de dados) também poderá permitir acesso aleatório ou apenas acesso sequencial, como um fluxo de dados baseado em `socket` (soquete). Por fim, a interface `Psr\Http\Message\StreamInterface` define um método `__toString()` para simplificar, recuperar ou emitir todo o conteúdo do corpo da mensagem de uma única vez.

Ao contrário das interfaces de requisição `Psr\Http\Message\RequestInterface` e resposta `Psr\Http\Message\ResponseInterface`, a interface `Psr\Http\Message\StreamInterface` não é um modelo de imutabilidade. Em casos em que um fluxo real do PHP é empacotado, a imutabilidade é impossível de ser aplicada, bem como em qualquer código que interage com o recurso, podendo alterar seu estado (incluindo posição do cursor, conteúdo etc.).

É recomendado que as aplicações utilizem fluxos somente leitura para requisições efetuadas do lado do servidor, e respostas do lado do cliente.

Além disso, as aplicações consumidoras DEVEM estar cientes de que a instância `Psr\Http\Message\StreamInterface` PODE ser mutável podendo alterar o estado da mensagem.

Regra 4

Solicitar destinos e URIs - As mensagens de solicitação

contêm um pedido-alvo como segundo segmento da linha de solicitação, sendo que o destino da solicitação pode ser alguma das seguintes formas:

- **`origin-form`** - Se presente na `string` de consulta, é chamado de URL (*Uniform Resource Locator*, ou Localizador Padrão de Recursos, em português) relativo. Geralmente, as mensagens transmitidas por TCP são de origem e os dados de esquema e autoridade geralmente só estão presentes por meio de variáveis CGI (*Common Gateway Interface*).
- **`absolute-form`** - Consiste no esquema, autoridade (`[user-info@]host[:port]` , onde os itens entre colchetes são opcionais), caminho (se presente), `string` de consulta (se presente) e fragmento (se presente). Isso geralmente é chamado de URI absoluto e é a única forma para especificar um URI. Essa forma é usada ao fazer solicitações para Proxies HTTP, por exemplo, em um ambiente corporativo em que existe proxy configurado, restringindo o acesso.
- **`authority-form`** - Consiste apenas na autoridade, é normalmente usado apenas em solicitações `CONNECT` para estabelecer uma conexão entre um cliente HTTP e um servidor de proxy.
- **`asterisk-form`** - Consiste unicamente na `string *` e é utilizado com o método `OPTIONS` para determinar as capacidades gerais de um determinado servidor da Web.

Além desses pedidos-alvo, ou metas de solicitação, há algumas vezes em que um URL efetivo pode ser encontrado separado da segmentação da solicitação.

Esse URL efetivo não é transmitido em uma mensagem HTTP, mas é utilizado para determinar o tipo de protocolo que será utilizado (`http/https`), a porta e o nome do host para fazer a requisição.

O URI efetivo é representado pela interface `Psr\Http\Message\UriInterface` que disponibiliza métodos para interagir com diversas partes do URI, o que eliminará a necessidade de análise repetitiva. Além disso, especifica um método `__toString()` para que seja possível converter o objeto URI em uma representação em formato de `string` .

Ao recuperar o pedido-alvo com o método `getRequestTarget()` , por padrão, o método utilizará o objeto URI e extrairá todos os componentes necessários para construir o formulário de origem - este é o pedido-alvo mais comum.

Caso seja necessário que um usuário final utilize uma das outras formas, ou se o usuário desejar substituir o destino da solicitação, é possível fazer essa alteração utilizando o método `withRequestTarget()` . A chamada desse método não vai afetar o URI, pois ele é retornado através do método `getUri()` .

Vamos analisar um exemplo em que o usuário efetua uma solicitação usando o formulário de asterisco para um determinado servidor:

```
<?php

namespace App\Middleware\Authorization

use Psr\Http\Message\RequestInterface;

class AuthorizationMiddleware
{
```

```
        private $request;

        public function __construct(RequestInterface $request)
        {
            $this->request = $request->withMethod('OPTIONS')
                ->withRequestTarget('*')
                ->withUri(new Uri('https://www.php-fig.org/'));
        }
    }
```

O resultado obtido dessa alteração seria semelhante ao mostrado a seguir:

```
OPTIONS * HTTP/1.1
```

Caso necessário, o cliente HTTP pode utilizar o URL efetivo obtido através do método `getUri()` para definir o protocolo, o hostname e a porta a ser utilizada.

O cliente HTTP DEVE ignorar os valores contidos nos métodos `getPath()` e `getQuery()` presentes na interface `Psr\Http\Message\UriInterface` e utilizar o valor obtido através do método `getRequestTarget()`, que por padrão concatena os valores obtidos nos métodos anteriormente citados (`getPath()` e `getQuery()`).

Os clientes que optarem por não fazer a implementação de uma ou mais das 4 formas de pedido-alvo, DEVERÃO, mesmo assim, utilizar o método `getRequestTarget()`.

Esses clientes DEVEM rejeitar as metas de solicitação porque eles não suportam e NÃO DEVEM retroceder nos valores obtidos através do método `getUri()`.

A interface `Psr\Http\Message\RequestInterface` disponibiliza métodos para recuperar a requisição ou criar uma

nova instância com a requisição fornecida.

Por padrão, caso o destino da solicitação não seja especificamente composto, então o método `getRequestTarget()` retornará o formulário de origem do URI composto, ou `/` se nenhum URI for composto.

O método `withRequestTarget($requestTarget)` é responsável por criar uma nova instância com a solicitação de destino especificada, permitindo que os desenvolvedores criem mensagens de requisição que possam representar as outras três formas de requisição de destino (formulário absoluto, formulário de autoridade e formato asterisco).

Quando utilizada, a instância de URI composta ainda pode ser utilizada, principalmente em clientes que requerem a criação da conexão com o servidor.

Regra 5

Requisições do lado do servidor - A interface `Psr\Http\Message\RequestInterface` fornece uma representação geral de uma requisição de uma mensagem HTTP, porém essas requisições precisam de um tratamento adicional do lado do servidor.

O processamento do lado do servidor precisa levar em consideração a CGI e ainda a abstração do PHP por meio da extensão de CGI.

O PHP fornece uma simplificação em torno do empacotamento de entrada por meio de variáveis superglobais, como:

- **`$_COOKIE`** - Desserializa e concede acesso simplificado aos `cookies` HTTP.
- **`$_GET`** - Desserializa e concede acesso simplificado aos parâmetros da string de consulta.
- **`$_POST`** - Desserializa e concede acesso simplificado aos parâmetros enviados via método `POST` do HTTP.
- **`$_FILES`** - Provê metadados que são serializados ao fazer uploads de arquivos.
- **`$_SERVER`** - Concede acesso a variáveis de ambiente CGI/SAPI, que incluem o método de requisição, o esquema de requisição, o URI de requisição e os cabeçalhos.

O QUE É SAPI (*SERVER APPLICATION PROGRAMMING INTERFACE*)? Server Application Programming Interface ou Interface de Programação de Aplicativos do Servidor em português, como o próprio nome sugere, é uma interface de programação de aplicativos que é fornecida pelo servidor, possibilitando que desenvolvedores estendam os recursos do servidor.

O que é CGI (*Common Gateway Interface*)? Common Gateway Interface ou Interface Comum de Porta de Entrada em português, é uma tecnologia que permite a geração de páginas dinâmicas através de parâmetros enviados pelo navegador a scripts que estão armazenados no servidor. Os scripts CGI interpretam os parâmetros, realizam o processamento e, por fim, geram a página.

A interface `Psr\Http\Message\ServerRequestInterface`

estende a interface `Psr\Http\Message\RequestInterface` para que seja possível obter uma abstração envolvendo essas variáveis superglobais, ajudando a reduzir o acoplamento junto a elas pelos clientes.

Além disso, incentiva e promove a capacidade de testar a solicitação do cliente. A solicitação do servidor fornece uma propriedade adicional chamada `attributes` para permitir aos clientes a capacidade de introspecção, decomposição e combinação da solicitação contra regras específicas da aplicação como: caminho, esquema, host etc. A solicitação do servidor também pode fornecer mensagens entre vários pedidos.

Upload de arquivos - A interface `Psr\Http\Message\ServerRequestInterface` especifica um método para recuperar uma árvore de arquivos decorrentes de uploads em uma estrutura onde cada folha é uma instância de `Psr\Http\Message\UploadFileInterface`.

É importante ressaltar que a variável superglobal `$_FILES` possui alguns problemas ao lidar com `arrays` de entradas de arquivos, por exemplo, se a aplicação enviar um `array` de arquivos contendo uma chave chamada `arquivos` e submetendo `$arquivos[0]` e `$arquivos[1]`, teremos a seguinte saída dada pelo PHP.

```
array(
    'arquivos' => array(
        'name' => array(
            0 => 'file0.txt',
            1 => 'file1.html',
        ),
        'type' => array(
            0 => 'text/plain',
            1 => 'text/html',
```

```
        ),
        /* etc. */
    ),
)
```

Quando, na verdade, o esperado seria algo como:

```
array(
    'arquivos' => array(
        0 => array(
            'name' => 'file0.txt',
            'type' => 'text/plain',
            //...
        ),
        1 => array(
            'name' => 'file1.html',
            'type' => 'text/html',
            //...
        ),
    ),
)
```

Os desenvolvedores precisam conhecer o detalhe dessa implementação da linguagem PHP e assim escrever um código que possa reunir os dados de um upload em específico. Além disso, há cenários em que `$_FILES` não é preenchido:

- Quando o método HTTP não é `POST` ;
- Quando estiver utilizando teste de unidade;
- Quando o ambiente não for SAPI.

Nesses casos, os dados terão que ser distribuídos de maneira diferente, por exemplo:

- Um processo pode tentar analisar o corpo da mensagem para descobrir os uploads de arquivos. Em alguns casos, a aplicação pode não gravar os uploads de arquivos no sistema de arquivos, mas pode colocá-los em um fluxo para

reduzir o uso da memória, E/S (Entrada e Saída) e armazenamento.

- Durante os testes de unidade, os desenvolvedores precisam simular os metadados de upload de arquivo para validar e verificar diferentes cenários.

O método `getUploadedFiles()` fornece uma estrutura padronizada para a aplicação, sendo que é esperado que:

- Todas as informações para um determinado upload de arquivo sejam agregadas e usadas para popular a instância de `Psr\Http\Message\UploadedFileInterface` .
- Ao recriar a estrutura da árvore, deve-se considerar que cada folha é uma instância de `Psr\Http\Message\UploadedFileInterface` para um local determinado na árvore.

A estrutura da árvore referenciada deve imitar a estrutura de nomenclatura à qual os arquivos foram submetidos. Veja o exemplo a seguir, que demonstra a estrutura utilizando `$_FILES` :

```
array(
    'php_field' => array(
        'tmp_name' => 'phpUelOru',
        'name' => 'php-logo.png',
        'size' => 10586,
        'type' => 'image/png',
        'error' => 0,
    ),
)
```

Agora veja como ficaria utilizando o método `getUploadedFiles()` :

```
array(
    'php_field' => //Instância de UploadFileInterface
```

```
    ),
)
```

Você pode definir um `array` de arquivos, mas a implementação da especificação deve agregar todas as informações relacionadas ao arquivo de acordo com o índice fornecido, isso porque `$_FILES` acaba saindo da sua estrutura convencional em alguns casos. Como os dados dos arquivos enviados são derivados de `$_FILES` ou do corpo do pedido, o método `withUploadedFiles()` também está disponível na interface `Psr\Http\Message\UploadedFileInterface`, permitindo a delegação da padronização para um outro processo. Além disso, a interface disponibiliza métodos que garantem que as operações funcionarão independentemente do ambiente:

- O método `moveTo($targetPath)` é disponibilizado como uma alternativa recomendada e segura para realizar chamadas através do método `move_uploaded_file()` diretamente no arquivo temporário. As implementações vão detectar a operação correta para ser utilizada conforme o ambiente.
- O método `getStream()` retornará uma instância de `Psr\Http\Message\StreamInterface`. Em ambientes que não sejam SAPI. A proposta é realizar uma análise dos arquivos de upload individuais em `php://temp`. Em alguns casos, o arquivo de upload não está presente, portanto, o método `getStream()` é garantido para manter a funcionalidade independentemente do ambiente.

8.2 INTERFACES

Após a passagem por toda a PSR-7, não podíamos deixar de

especificar as interfaces que fazem parte desse pacote, então vamos conferi-las a seguir:

Psr\Http\Message\MessageInterface

```
<?php

namespace Psr\Http\Message;

interface MessageInterface
{
    public function getProtocolVersion();
    public function withProtocolVersion($version);
    public function getHeaders();
    public function hasHeader($name);
    public function getHeader($name);
    public function getHeaderLine($name);
    public function withHeader($name, $value);
    public function withAddedHeader($name, $value);
    public function withoutHeader($name);
    public function getBody();
    public function withBody(StreamInterface $body);
}
```

Psr\Http\Message\RequestInterface

```
<?php

namespace Psr\Http\Message;

interface RequestInterface extends MessageInterface
{
    public function getRequestTarget();
    public function withRequestTarget($requestTarget);
    public function getMethod();
    public function withMethod($method);
    public function getUri();
    public function withUri(UriInterface $uri, $preserveHost = fa
lse);
}
```

Psr\Http\Message\ServerRequestInterface

```php
<?php

namespace Psr\Http\Message;

interface ServerRequestInterface extends RequestInterface
{
    public function getServerParams();
    public function getCookieParams();
    public function withCookieParams(array $cookies);
    public function getQueryParams();
    public function withQueryParams(array $query);
    public function getUploadedFiles();
    public function withUploadedFiles(array $uploadedFiles);
    public function getParsedBody();
    public function withParsedBody($data);
    public function getAttributes();
    public function getAttribute($name, $default = null);
    public function withAttribute($name, $value);
    public function withoutAttribute($name);
}
```

Psr\Http\Message\ResponseInterface

```php
<?php

namespace Psr\Http\Message;

interface ResponseInterface extends MessageInterface
{
    public function getStatusCode();
    public function withStatus($code, $reasonPhrase = '');
    public function getReasonPhrase();
}
```

Psr\Http\Message\StreamInterface

```php
<?php

namespace Psr\Http\Message;

interface StreamInterface
{
```

```
    public function __toString();
    public function close();
    public function detach();
    public function getSize();
    public function tell();
    public function eof();
    public function isSeekable();
    public function seek($offset, $whence = SEEK_SET);
    public function rewind();
    public function isWritable();
    public function write($string);
    public function isReadable();
    public function read($length);
    public function getContents();
    public function getMetadata($key = null);
}
```

Psr\Http\Message\UriInterface

```
<?php

namespace Psr\Http\Message;

interface UriInterface
{
    public function getScheme();
    public function getAuthority();
    public function getUserInfo();
    public function getHost();
    public function getPort();
    public function getPath();
    public function getQuery();
    public function getFragment();
    public function withScheme($scheme);
    public function withUserInfo($user, $password = null);
    public function withHost($host);
    public function withPort($port);
    public function withPath($path);
    public function withQuery($query);
    public function withFragment($fragment);
    public function __toString();
}
```

Psr\Http\Message\UploadedFileInterface

```
<?php

namespace Psr\Http\Message;

interface UploadedFileInterface
{
    public function getStream();
    public function moveTo($targetPath);
    public function getSize();
    public function getError();
    public function getClientFilename();
    public function getClientMediaType();
}
```

Conclusão

Vimos a PSR-7, que define interfaces em comum para que a troca de mensagens HTTP se torne mais simples. Não é uma regra tão simples de ser seguida e é preciso ter muita atenção no momento em que estiver implementando-a em sua aplicação. São muitos detalhes que devem ser seguidos para que o seu código esteja em conformidade com essa PSR.

No próximo capítulo vamos conhecer a PSR-11, suas regras e seus conceitos.

Capítulo 9

PSR-11 (CONTAINER INTERFACE)

A PSR-11 especifica uma interface comum para a injeção de dependência através de contêineres (recipientes). Atualmente muitos frameworks já a utilizam, um bom exemplo é o próprio Zend Framework e o Zend Expressive. Não é uma PSR complexa e, diferentemente da PSR-7, essa PSR é bem simples e fácil.

9.1 REGRAS

A seguir, vamos conhecer os elementos que descrevem a PSR-11, assim como suas regras e suas interfaces.

Regra 1

Identificadores de entrada - Pode ser considerada como identificador de entrada qualquer sequência de caracteres que identifica exclusivamente um item contido em um contêiner. Nada mais é que uma string opaca, portanto, os chamadores NAO DEVEM presumir que a estrutura da string carrega qualquer significado semântico.

Regra 2

Lendo de um contêiner - A interface `Psr\Container\ContainerInterface` expõe dois métodos: `get()` e `has()`.

O método `get()` possui apenas um parâmetro, que é obrigatório. Esse parâmetro DEVE ser uma string que é o nome do identificador.

Esse método pode retornar qualquer valor ou até mesmo lançar uma exceção do tipo `Psr\Container\NotFoundExceptionInterface` caso o identificador passado como parâmetro do método não seja reconhecido pelo contêiner.

Duas chamadas sucessivas para obter o mesmo identificador DEVEM retornar exatamente o mesmo valor. Isso significa que, se você tiver um contêiner de serviços chamado `usuario` e realizar 1, 2, ou N chamadas para este contêiner, o valor retornado pelo contêiner deve ser exatamente o mesmo.

O método `has()` também possui somente um parâmetro, que é um identificador, sendo que este DEVE ser uma string. O método DEVE retornar `true` se o identificador for reconhecido pelo contêiner e `false` caso não seja. Caso o método `has()` retorne `false`, então o método `get()` DEVE lançar uma exceção do tipo `Psr\Container\NotFoundExceptionInterface`.

Regra 3

Exceções - As exceções que são lançadas diretamente pelo contêiner DEVEM implementar a interface

`Psr\Container\ContainerExceptionInterface`. Caso o ID passado no método `get()` seja inexistente, então DEVE ser lançada uma exceção do tipo `Psr\Container\NotFoundExceptionInterface`.

Regra 4

Uso recomendado - Os desenvolvedores NÃO DEVEM passar um contêiner em um objeto para que este objeto possa recuperar suas próprias dependências. Isso significa que o contêiner é usado como um localizador de serviço (`Service Locator`), ou seja, ele localizará ou não um determinado serviço através do nome informado.

Se o nome informado existir, então pode-se retornar a instância do objeto pertencente ao serviço. Por exemplo, temos um contêiner chamado `usuario` que retorna a instância de um serviço chamado `Usuario`.

9.2 INTERFACES

Agora que conhecemos as regras para sua utilização, vamos conhecer as três interfaces que fazem parte desse pacote.

`Psr\Container\ContainerInterface`

```php
<?php

namespace Psr\Container;

interface ContainerInterface
{
    public function get($id);
    public function has($id);
}
```

Psr\Container\ContainerExceptionInterface

```
<?php

namespace Psr\Container;

interface ContainerExceptionInterface
{
}
```

Psr\Container\NotFoundExceptionInterface

```
<?php

namespace Psr\Container;

interface NotFoundExceptionInterface extends ContainerExceptionIn
terface
{
}
```

Conclusão

Como você pôde perceber, essa PSR define uma única interface usada para injeção de dependências através de contêineres. Ela é bem simples, possui apenas dois métodos, `get()` e `has()`, que são utilizados para obter e verificar as informações respectivamente.

No próximo capítulo falaremos sobre a PSR-12, que é uma extensão da PSR-2 e que possui novas regras de formatação compatíveis com as novas funcionalidades do PHP 7, então vamos nessa!

Capítulo 10

PSR-12 (EXTENDED CODING STYLE) - PARTE I

A PSR-12 é uma extensão da PSR-2 e possui o mesmo objetivo que ela, porém com um contexto mais moderno e novas funcionalidades que seguem a evolução do PHP. Se pararmos para pensar, a PSR-2 foi aceita em 2012 e desde então vinha sofrendo atualizações para acompanhar a evolução do PHP e fazer com que o código pudesse ser interpretado de maneira mais eficaz entre os desenvolvedores. Como o PHP 7 teve muitas mudanças, a reescrita da PSR-2 em uma nova PSR foi necessária para atender a todas as novidades disponíveis no PHP 7.

Veremos na seção a seguir quais são as regras, os conceitos e como aplicá-los em seu projeto para que se obtenha um código mais bonito e legível compatível com o PHP 7 e suas funcionalidades.

Vale ressaltar que a PSR-2 foi descontinuada, mas não foi removida deste livro por questões didáticas e de entendimento, pois assim você poderá notar as diferenças entre ambas, PSR-2 e PSR-12.

10.1 REGRAS

Confira nesta seção todas as regras e características dessa PSR.

Regra 1

Os códigos PHP DEVEM seguir os padrões estabelecidos na PSR-1 (`Basic Code Standard`). Por ser uma extensão da PSR-1, seu código primeiramente deve estar em conformidade com a PSR-1 antes de aplicar a PSR-2.

O termo `StudlyCaps` no PSR-1 DEVE ser interpretado como `PascalCase` , onde a primeira letra de cada palavra é maiúscula.

Regra 2

Arquivos PHP DEVEM possuir o padrão UNIX de terminação de linha. Você sabe quais são as terminações de linhas disponíveis? Caso não, vejamos a seguir:

- **CRLF**: retorno de carro + avanço de linha, Unicode caracteres D + 000A 000
- **LF**: alimentação de linha, caracteres Unicode 000A
- **NEL**: próxima linha, o caractere Unicode 0085
- **LS**: separador de linha, caractere Unicode 2028
- **PS**: separador de parágrafo, caractere Unicode 2029

O padrão UNIX adotado pela PSR-12 é o LF. Geralmente, as IDEs já vêm configuradas para o padrão UNIX, mas isso pode ser alterado facilmente dependendo da IDE utilizada.

Regra 3

Arquivos contendo apenas código PHP DEVEM possuir uma linha em branco ao final. Após toda a lógica aplicada, deve haver

uma linha em branco ao final do arquivo. Veja o exemplo a seguir, que demonstra a aplicação dessa regra:

```
1.<?php
2.class Usuario
3.{
4.    private $nome;
5.
6.    public function __construct($nome)
7.    {
8.        $this->nome = $nome;
9.    }
10.}
11.//Linha em branco, sem espaços
```

Regra 4

Arquivos contendo apenas PHP DEVEM ter a TAG de fechamento omitida. Pode parecer estranho no primeiro momento, mas o seu script PHP não vai deixar de funcionar se a TAG de fechamento (`?>`) for omitida. A regra não se aplica aos arquivos contendo HTML + PHP. Para facilitar o entendimento, vamos ver o exemplo a seguir que demonstra a forma correta de aplicar essa regra:

```
<?php
class Usuario
{
    private $nome;

    public function __construct($nome)
    {
        $this->nome = $nome;
    }
}
```

Perceba que a TAG de fechamento (`?>`) foi omitida e isso não afeta o funcionamento do código. Agora vamos analisar o exemplo em que a omissão da TAG ocasionará em erro no código:

```
<?php
$nome = 'Jhones S. Clementino';

<html>
    <head>
        <title>Teste</title>
    </head>
    <body>
        <h1>Olá, <?php echo $nome</span>!</h1>
    </body>
</html>
```

Diferentemente do exemplo anterior, a omissão da TAG de fechamento do PHP (`?>`) ocasionará erro após a declaração da variável `$nome` , isso porque o arquivo não possui apenas código PHP.

Regra 5

O comprimento máximo de uma linha de código DEVE ser de 120 caracteres. Verificadores de estilo automatizado DEVEM avisar, mas NÃO DEVEM exibir erro.

As linhas de código NÃO DEVEM ter mais de 80 caracteres. Linhas de código mais longas DEVEM ser quebradas em várias linhas subsequentes de no máximo 80 caracteres.

Pode parecer estranha a explicação, mas faz todo o sentido. Em sistemas legados, principalmente, há linhas com mais de 1000 caracteres de comprimento - acredite, não é exagero! Já trabalhei em sistemas desse tipo e é complicado dar manutenção nesse tipo de código.

Portanto, atente-se para que a cada 80 caracteres você possa quebrar a linha de modo que fique legível e, caso não consiga, você ainda possui o limite de 120 caracteres.

Não é um erro, mas se quiser seguir o padrão da PSR-12 e manter suas boas práticas é muito importante atentar-se a mais esta regra.

Regra 6

Linhas que possuem conteúdo NÃO DEVEM possuir espaços em branco ao final de seu conteúdo.

Regra 7

Para facilitar a interpretação do código, linhas em branco PODEM ser adicionadas. Você pode adicionar linhas em branco para tornar a interpretação do código mais fácil. Vamos ver a seguir um breve exemplo da aplicação dessa regra:

```
<?php

class Usuario
{
    private $nome;

     private $idade;

     private $sexo;

    public function __construct($nome, $idade, $sexo)
    {
          $this->nome = $nome;
         $this->idade = $idade;

         if ($sexo == 'M') {
            $this->sexo = 'Masculino';
        } elseif ($sexo == 'F') {
            $this->sexo = 'Feminino';
        }
    }
}
```

Perceba como o código ficou legível com a aplicação das quebras de linha. Agora veja o exemplo a seguir, em que não existe quebra de linha na escrita da lógica:

```
<?php

class Usuario
{
    private $nome;
     private $idade;
     private $sexo;
    public function __construct($nome, $idade, $sexo)
    {
          $this->nome = $nome;
         $this->idade = $idade;
         if ($sexo == 'M') {
            $this->sexo = 'Masculino';
        } elseif ($sexo == 'F') {
            $this->sexo = 'Feminino';
        }
    }
}
```

O código anterior funciona sem nenhum problema, mas perceba como fica um pouco estranho e de certa forma um pouco difícil de entender principalmente a separação entre os atributos da classe e o método construtor.

Regra 8

NÃO DEVE haver múltiplas declarações em uma única linha. Cada declaração de variável deve ser feita em uma nova linha. Ou seja, definiu uma variável? Então pule uma linha para definir a próxima variável.

Vamos ver o exemplo a seguir que demonstra a maneira considerada correta ao declarar uma variável seguindo o padrão da PSR-12:

```php
<?php
$nome = 'Jhones';
$sobrenome = 'S. Clementino';
```

Está correto, do jeito que tem que ser, uma declaração por linha. Agora veremos a maneira considerada incorreta:

```php
<?php
$nome = 'Jhones'; $sobrenome = 'S. Clementino';
```

Este é um exemplo bem simples que não é considerado correto pela PSR-12, então lembre-se de declarar uma variável, constante, função etc. por linha.

Regra 9

Códigos PHP DEVEM possuir indentação com 4 espaços e não com TABs. Não é uma regra complexa, basta atentar-se aos 4 espaços e em vez do costume de utilizar TABs.

Algumas IDEs já vêm com o padrão de 4 espaços ao teclar o TAB, mas vale conferir para que seu código não saia das conformidades da PSR-12. Isso pode ser configurado sem muita dor de cabeça.

Regra 10

As palavras-chaves, tipos reservados e constantes `true`, `false` e `null` DEVEM ser escritas com letras minúsculas. Simplesmente use-as com letras minúsculas e não use variações como: `TRUE`, `FALSE`, `NULL` ou `True`, `False`, `Null`, ou `TrUe`, `FaLsE`, `NuLl`.

Quaisquer novos tipos e palavras-chaves adicionados em versões futuras do PHP DEVEM estar em minúsculas.

É necessário usar uma forma abreviada de palavras-chaves do tipo de dado, ou seja, `bool` em vez de `boolean`, `int` em vez de `integer` etc.

Regra 11

O cabeçalho de um arquivo PHP pode consistir em vários blocos diferentes. Se o cabeçalho estiver presente, cada um dos blocos a seguir DEVE ser separado por uma única linha em branco e NÃO DEVE conter uma linha em branco.

Cada bloco DEVE estar na ordem listada a seguir, mas os blocos que não são relevantes podem ser omitidos.

- Tag de abertura `<?php` do PHP;
- Docblock `/** Descrição */` no nível do arquivo;
- Uma ou mais declarações `declare()`;
- A definição do `namespace` do arquivo;
- Uma ou mais instruções de importação `use` baseadas em classes;
- Uma ou mais instruções de importação `use` baseadas em funções;
- Uma ou mais instruções de importação `use` baseadas em constantes;
- O restante do código.

Quando um arquivo contém HTML e PHP, qualquer uma das seções descritas anteriormente ainda poderá ser utilizada. Nesse caso, eles DEVEM estar presentes no topo do arquivo, mesmo que o restante do código consista em uma tag de fechamento `?>` do PHP e, em seguida, uma mistura de HTML e PHP.

As instruções de importação nunca DEVEM começar com uma barra invertida.

Vamos conferir um exemplo a seguir que ilustra a aplicação da sequência definida pela PSR-12 citada anteriormente:

```
<?php

/**
 * Descrição do arquivo
 */

 declare(strict_types=1);

 namespace App\Controller\Usuario;

 use App\Controller\{AbstractController as Abc, ControllerInterfa
ce, BaseController as Base};
 use App\Entity\Usuario

 use function App\Functions\{functionA, functionN, functionC};
 use function App\Repository\Functions\insert;

 use const App\Controller\{RESPONSE_JSON, RESPONSE_XML, RESPONSE_
ERROR};
 use const App\Service\{CONSTANT_A}

/**
 *  Classe de exemplo
 */
 class UsuarioController
 {
     //Código...
 }
```

Uau, perceba como tivemos mudanças, se compararmos com a PSR-2. Uma delas é a importação de múltiplas classes dentro de uma mesma instrução `use` através da utilização das chaves `{}`. Se você recordar, na PSR-2 era permitido apenas uma importação por instrução `use`.

Outras evoluções foram com relação à importação de funções e constantes da mesma maneira que importamos uma ou mais classes. Sem dúvidas, foi uma bela evolução e que com certeza nos ajudará muito a manter o código com um padrão bem definido.

Regra 12

Os namespaces compostos com profundidade superior a dois NÃO DEVEM ser utilizados. O exemplo a seguir demonstra a utilização máxima de composição permitida:

```
<?php

use App\Service\{
    Usuario\UsuarioService,
    Contato\ContatoService,
    ServiceAbstract,
}
```

O exemplo a seguir não é permitido pela PSR-12:

```
<?php

use App\Service\{
    Usuario\Admin\UsuarioService,
    Contato\ContatoService,
    ServiceAbstract,
}
```

Nesse exemplo, a adição de mais um `subnamespace` no caso `Admin` não é permitido, pois ultrapassa o nível máximo de profundidade permitido, que é até dois níveis.

Regra 13

Se desejar declarar tipos estritos em arquivos contendo marcação HTML fora das TAGs de abertura `<?php` e fechamento

`?>` do PHP, a declaração DEVE estar na primeira linha do arquivo e incluir uma TAG de abertura `<?php` do PHP, a declaração de tipos estritos e em seguida TAG de fechamento `?>` do PHP.

Vamos conferir o exemplo a seguir que demonstra a aplicação dessa regra:

```
<?php declare(strict_types=1) ?>
<html>
<body>
    <?php //Código PHP... ?>
</body>
</html>
```

Instruções `declare()` NÃO DEVEM conter espaços e DEVEM ser declaradas exatamente da maneira a seguir: `declare(strict_types=1)` . O ponto e vírgula no final da instrução é opcional.

Regra 14

As declarações de bloco `declare` contendo o código a ser executado são permitidas e DEVEM ser formatadas como demonstrado no exemplo a seguir. Observe a posição das chaves e o espaçamento:

```
<?php
declare(ticks=1) {
    // código...
}
```

Regra 15

Após qualquer chave de fechamento, NÃO DEVE haver nenhum comentário ou declaração na mesma linha.

Ao instanciar uma nova classe, os parênteses DEVEM estar sempre presentes, mesmo quando não houver parâmetros passados para o construtor.

Vamos ver o exemplo a seguir que demonstra a aplicação dessa regra:

```
<?php

new Usuario();
```

Regra 16

As palavras-chaves `extends` e `implements` DEVEM ser declaradas na mesma linha que contém o nome da classe.

A chave de abertura `{` da classe DEVE ser inserida em uma nova linha e NÃO DEVE ser precedida ou seguida por uma linha em branco.

A chave de fechamento `}` da classe também DEVE ser inserida em uma nova linha logo após a definição do corpo da classe e NÃO DEVE ser precedida ou seguida por uma linha em branco.

Vamos ver o exemplo a seguir, que demonstra a aplicação dessa regra:

```
<?php

namespace App\Controller\Usuario;

use App\Controller\ControllerAbstract;
use App\Service\UsuarioService;

class UsuarioController extends ControllerAbstract implements \Se
rializable, \Countable
```

```
{
    //Definição do corpo da classe...
}
```

Regra 17

Quando houver a necessidade de múltiplos `implements` , eles PODEM ser divididos em múltiplas linhas, sendo que cada linha subsequente é recuada uma vez. Sempre que fizer isso, o primeiro item da lista DEVE estar na próxima linha e DEVE possuir somente uma interface por linha.

Se você ainda nunca viu ou nunca ouviu falar em uma classe implementando múltiplas interfaces, saiba que não é incorreto e, sim, existem muitos projetos que, devido à complexidade, acabam implementando esse tipo de lógica.

Vamos analisar o exemplo a seguir, que demonstra a aplicação correta dessa regra:

```
<?php

namespace App\Controller\Usuario;

use App\Controller\ControllerAbstract;
use App\Service\UsuarioService;

class UsuarioController extends ControllerAbstract implements
    \Serializable,
    \Countable
{
    //Definição do corpo da classe...
}
```

O exemplo demonstra com clareza exatamente o que a PSR-12 define, ou seja, quando há múltiplas interfaces, cada interface deve estar em uma nova linha.

Regra 18

Para utilizar `Traits` e usufruir da herança vertical dentro das classes, a instrução `use` DEVE ser declarada na próxima linha após a chave `{` de abertura da classe.

Veja o exemplo a seguir, que demonstra a aplicação dessa pequena regra:

```
<?php

namespace App\Controller\Usuario;

use App\Usuario\UsuarioTrait;

class UsuarioController
{
    use UsuarioTrait;
}
```

Regra 19

Cada `Trait` a ser utilizada dentro da classe DEVE ser incluída uma por linha, sendo que cada `Trait` DEVE ter a sua própria declaração de uso `use` .

Veja o exemplo a seguir para melhor entendimento:

```
<?php

namespace App\Controller\Usuario;

use App\Usuario\UsuarioTrait;
use App\Contato\ContatoTrait;
use App\Endereco\EnderecoTrait;

class UsuarioController
{
    use UsuarioTrait;
    use ContatoTrait;
```

```
    use EnderecoTrait;
}
```

Regra 20

Quando a classe não tem nada após a declaração de uso `use`, a chave de fechamento da classe DEVE estar na linha subsequente.

Já vimos isso nos exemplos anteriores de `Traits`, mas para exemplificar vamos verificar o exemplo a seguir:

```
<?php

namespace App\Controller\Usuario;

use App\Usuario\UsuarioTrait;

class UsuarioController
{
    use UsuarioTrait;
}
```

Mas e se a classe tiver conteúdo após a declaração de uso da `Trait`? Então é simples, DEVE ter uma linha em branco após a declaração de uso `use`, conforme demonstra o exemplo a seguir:

```
<?php

namespace App\Controller\Usuario;

use App\Usuario\UsuarioTrait;

class UsuarioController
{
    use UsuarioTrait;

    private $usuario;
}
```

Regra 21

O uso dos operadores `insteadof` e `as` deve ser feito da seguinte maneira, percebendo recuo, espaçamento e novas linhas:

```
<?php

namespace App\Controller\Usuario;

use App\Usuario\UsuarioTrait;
use App\Contato\ContatoTrait;
use App\Endereco\EnderecoTrait;

class UsuarioController
{
    use A, B, C {
        B::getUsuario insteadof A;
        A::getContato insteadof B;
        C::getEndereco as Endereco;
    }
}
```

Regra 22

A visibilidade ou modificadores de acesso DEVEM ser definidos em todas as propriedades. Toda propriedade que for definida deve possuir o modificador de acesso definido antes do seu nome.

O exemplo a seguir demonstra como é a aplicação dessa regra:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    private $sobrenome;

    protected static $sexo;
```

```
    public int $idade;

    //Mais propriedades, métodos etc...
}
```

Perceba que todas as propriedades que foram definidas possuem antes de seu nome o modificador de acesso, seja ele `private`, `protected` ou `public`.

Agora vamos analisar o mesmo exemplo, porém sem os modificadores de acesso:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    $nome;

    $sobrenome;

    static $sexo;

    int $idade;
}
```

Analisando o código do exemplo anterior, fica difícil de saber qual propriedade é privada, pública ou protegida.

Na verdade, quando o modificador de acesso não é definido na propriedade, um erro chamado **PHP Parse error** será lançado, pois dentro de uma classe obrigatoriamente as suas propriedades devem possuir o modificador de acesso, com exceção das propriedades estáticas `static`, que automaticamente assumirão a visibilidade pública.

Regra 23

A palavra-chave `var` NÃO DEVE ser utilizada para a definição de propriedades da classe. No momento de definir as propriedades da classe, não utilize `var`, ou seja, o exemplo a seguir não é considerado correto pela PSR-12:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    var $nome;

    var $sobrenome;

    var $sexo;

    var $idade;
}
```

Apesar de não ser correto, o código será interpretado corretamente, e automaticamente todas as propriedades assumirão a visibilidade pública.

Regra 24

A visibilidade DEVE ser declarada em todas as constantes se a versão mínima do PHP em seu projeto for 7.1 ou superior, caso contrário, a regra não se aplica.

Vamos analisar o exemplo a seguir, que demonstra a aplicação dessa novidade:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
```

```
    private const RG = '112341233';

    protected const CPF = '111.222.333-44';

    public const NOME = 'Jhones S. Clementino';
}
```

Se você já desenvolveu utilizando as versões anteriores do PHP, sabe que ao definir as constantes não era possível informar a visibilidade de cada uma, porque o próprio PHP assumia automaticamente que cada constante definida possuía visibilidade pública.

Regra 25

NÃO DEVE haver mais de uma propriedade por declaração. Se houver a necessidade de declarar mais de uma propriedade, então defina-as de forma separada, conforme demonstra o exemplo a seguir:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    private $sobrenome;

    protected static $sexo;

    public int $idade;
}
```

Esse é o mesmo exemplo citado anteriormente. Como pode ser visto, cada declaração de propriedade está definida de forma separada. Agora veremos a maneira incorreta de se declarar as

propriedades da classe:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome, $sobrenome, $sexo, $idade;
}
```

Nesse exemplo, a declaração das propriedades está incorreta, porém, se você testar o código funcionará perfeitamente. Mas lembre-se: o código não estará em conformidade com a PSR-12.

Regra 26

O nome das propriedades NÃO DEVE iniciar com `_` (underscore) para indicar a visibilidade `private` ou `protected`. Certamente você já deve ter visto em algum lugar que há programadores que definem as propriedades privadas ou protegidas da classe com um `_` como prefixo.

Isso não é necessário, uma vez que o PHP fornece os modificadores de acesso, basta você utilizá-los. Vamos analisar a maneira correta ao definir as propriedades privadas ou protegidas de uma classe:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    private $sobrenome;
```

```
    protected static $sexo;

    public int $idade;
}
```

Perceba que é o mesmo exemplo que já fizemos anteriormente e não há segredo algum, o nome das propriedades está limpo e claro como deve ser. Agora vamos analisar um exemplo considerado incorreto:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $_nome;

    private $_sobrenome;

    protected static $_sexo;

    public int $idade;
}
```

Nesse exemplo, o modificador de acesso já informa o tipo de visibilidade da propriedade tornando completamente inútil o uso do _ (underscore).

Regra 27

DEVE haver um espaço entre a declaração de tipo e o nome da propriedade, já vimos isso nos exemplos anteriores, mas se você ficou na dúvida, pode conferir mais uma vez o código a seguir:

```
<?php

namespace App\Entity\Usuario;

class Usuario
```

```
{
    private $nome;

    private $sobrenome;

    protected static $sexo;

    public int $idade;
}
```

Perceba que, após a declaração do tipo `int`, temos um espaço e em seguida o nome da propriedade, que no caso é `$idade`.

Regra 28

Todos os métodos DEVEM ter a visibilidade definida. Lembre-se de que a visibilidade torna o encapsulamento mais claro. Para facilitar, vamos ao exemplo a seguir:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    public function getNome()
    {
        return $this->nome;
    }

    public function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

Perceba que os métodos `getNome()` e `setNome($nome)` possuem o modificador de acesso `public` definido, indicando

que eles possuem visibilidade pública.

O tipo de visibilidade vai depender do que será necessário expor ou não para o resto da aplicação. Agora vamos analisar o exemplo considerado incorreto e que não está em conformidade com a PSR-12:

```
<?php

namespace App\Entity\Usuario;

class Usuario
{
    private $nome;

    function getNome()
    {
        return $this->nome;
    }

    function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

É importante ressaltar que o código do exemplo funciona e automaticamente o PHP assumirá que os métodos possuem visibilidade pública, porém, não é válido para a PSR-12.

Regra 29

Após o nome do método NÃO DEVE existir um espaço em branco. A chave de abertura do método DEVE estar em uma nova linha e a chave de fechamento DEVE ficar logo após o corpo do método em uma nova linha.

Após o parêntese de abertura NÃO DEVE existir um espaço

em branco, e antes do parêntese de fechamento também NÃO DEVE haver um espaço em branco.

Pode parecer estranho, mas isso é considerado mais uma questão de adaptação e costume. Constantemente vejo códigos que não seguem essa regra da PSR. Para facilitar o entendimento, vamos ao exemplo considerado correto:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction()
    {
        //Código
    }
}
```

No exemplo citado, veja que o método `indexAction()` segue perfeitamente essa regra, ou seja, não há espaços após o nome do método, a chave de abertura está em uma nova linha e a chave de fechamento está logo após o conteúdo do método em uma nova linha. O mesmo se aplica para funções.

Vamos analisar agora um exemplo em que a definição do método não segue o padrão estipulado:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction ( $request ) {
        //Código
    }
}
```

Apesar de o código funcionar sem problemas, ele não é compatível com a regra da PSR-12.

Regra 30

Quando o método possuir uma lista de parâmetros, NÃO DEVE haver espaço antes de cada vírgula, mas DEVE ter espaço após cada vírgula. Sempre que você for definir uma lista de parâmetros no método, você deve separá-los por vírgula (,) e em seguida espaço, seguindo basicamente o padrão já usado em nosso próprio idioma.

Vamos ao exemplo considerando a maneira correta de definir os parâmetros do método:

```
<?php

namespace App\Controller;

class HomeController
{
    public function indexAction($request, $response)
    {
        //Código
    }
}
```

O exemplo mostra exatamente como a regra explica, cada parâmetro é separado por vírgula seguida de um espaço (,). A mesma regra se aplica para as funções. Agora vamos ver um exemplo que é considerado a maneira incorreta na definição dos parâmetros de um método/função:

```
<?php

namespace App\Controller;
```

```
class HomeController
{
    public function indexAction($request ,$response)
    {
        //Código
    }
}
```

Perceba que dessa vez os parâmetros são separados apenas pela vírgula, porém o espaço que deveria estar após a vírgula está definido antes da vírgula. Sendo assim, o código não está em conformidade com a PSR-12.

Regra 31

Caso o método possua muitos parâmetros a serem definidos, a lista de parâmetros PODE ser dividida em múltiplas linhas contendo uma indentação, sendo que o primeiro parâmetro da lista DEVE estar na próxima linha.

DEVE haver apenas um parâmetro por linha e o parêntese de fechamento da lista e a chave de abertura do método DEVEM estar na mesma linha, contendo apenas um espaço entre eles. A definição dessa regra pode ser grande mas não é difícil. Na verdade, a primeira vez que comecei a usar lá na PSR-2 eu estranhei bastante a forma como ficava o código, mas com o tempo acabei me acostumando.

A explicação desta regra fica mais fácil ao visualizar um exemplo que atenda à especificação, vamos nessa:

```
<?php

declare(strict_types=1);

namespace App\Sercice\Paginacao;
```

```php
class PaginacaoService
{
    public function paginate(
        int $limite,
        int $pagina,
        array $dados
    ): array {
        //Código
    }
}
```

Perceba como a lista de parâmetros foi dividida, ficou fácil de ler e identificar os parâmetros. Essa regra é muito bem aplicada em empresas que trabalhei, pois acredite, havia métodos que continham mais de 15 parâmetros, uma verdadeira aberração da programação.

Agora vamos analisar o mesmo exemplo em que a divisão em múltiplas linhas não é correta e deve ser evitada:

```php
<?php

declare(strict_types=1);

namespace App\Sercice\Paginacao;

class PaginacaoService
{
    public function paginate(
        int $limite, int $pagina,
        array $dados
    ): array {
        //Código
    }
}
```

O código do exemplo funciona perfeitamente, porém não está em conformidade com a regra de ter cada parâmetro em uma nova linha, então evite esse tipo de estilo.

Regra 32

Quando o(s) método(s) possuem a declaração de tipo de retorno definida, DEVE haver um espaço após os dois pontos seguido pela declaração de tipo. Os dois pontos e a declaração DEVEM estar na mesma linha que a lista de parâmetros entre parênteses, sem espaços entre os dois caracteres. O exemplo a seguir demonstra como é a aplicação dessa regra:

```
<?php

declare(strict_types=1);

namespace App\Sercice\Paginacao;

class PaginacaoService
{
    public function paginate(int $limite, int $pagina, array $dad
os): array
    {
        //Código
        return [];
    }

    public function getPagina(): int
    {
        return 1;
    }
}
```

Perceba que após o parêntese de fechamento `)` dos parâmetros do método `getPaginate` os dois pontos `:` foram definidos, indicando que o método obrigatoriamente deve ter um tipo de retorno definido, que nesse caso é o retorno do tipo `array`. Perceba também que entre os dois pontos `:` e o tipo de retorno `array` existe um espaço separando os dois.

Vamos analisar a seguir como seria o mesmo exemplo, porém

com a forma incorreta na formatação:

```
<?php

declare(strict_types=1);

namespace App\Sercice\Paginacao;

class PaginacaoService
{
    public function paginate(int $limite, int $pagina, array $dad
os):array
    {
        //Código
        return [];
    }

    public function getPagina():int
    {
        return 1;
    }
}
```

Esse exemplo é considerado incorreto porque não existe espaço entre os dois pontos `:` e o tipo de retorno de ambos os métodos.

Regra 33

Nas declarações de tipo `null`, NÃO DEVE haver um espaço entre o ponto de interrogação e o tipo. Essa regra parece estranha, mas não é e na verdade é bem simples. Vamos analisar o exemplo a seguir que demonstra como deve ser aplicada essa regra:

```
<?php

declare(strict_types=1);

namespace App\Entity\Usuario;

class Usuario
{
```

```
    public function getNome(): ?string
    {
        return $this->nome;
    }

    public function getDataAtualizacao(): ?\DateTime
    {
        return $this->dataAtualizacao;
    }

    public function setDados(?string $nome, ?int $idade): Usuario
    {
        $this->nome = $nome;
        $this->idade = $idade;
        return $this;
    }
}
```

Perceba que após os dois pontos `:` temos um espaço seguido por um ponto de interrogação `?` seguido pelo tipo de retorno. Essa sequência está informando que o método espera o retorno do tipo especificado ou um valor nulo `null`, é isso que o ponto de interrogação `?` significa, ou seja, ou retorna o valor esperado ou retorna nulo `null`.

Ainda no exemplo, perceba o método `setDados`. Nele, passamos dois parâmetros `?string $nome` e `?int $idade`, que podem ou não ser informados. Caso sejam informados, devem ser do tipo `string` e `int` respectivamente, mas caso não sejam informados, o valor nulo `null` será assumido. Perceba ainda que definimos que obrigatoriamente o retorno do método deve ser a instância da própria classe, no caso `Usuario`.

Regra 34

Ao usar o operador de referência e antes de um parâmetro, NÃO DEVE haver um espaço depois dele. NÃO DEVE haver um

espaço entre o operador variável de três pontos `...` e o nome do parâmetro.

Vamos analisar o exemplo que demonstra a aplicação dessa regra:

```
<?php

declare(strict_types=1);

namespace App\Crypt;

class Crypt
{
    public function crypt(string $string, ...$parts): string
    {
        //Código
    }
}
```

Perceba que na definição do segundo parâmetro nós temos o operador variável de três pontos `...` seguido pelo nome do parâmetro `$parts`. Ao utilizar esse operador, não deve haver espaço entre os três pontos `...` e o nome do parâmetro.

Vamos analisar um exemplo que é a forma incorreta de utilizar o operador de três pontos `...`:

```
<?php

declare(strict_types=1);

namespace App\Crypt;

class Crypt
{
    public function crypt(string $string, ... $parts): string
    {
        //Código
    }
}
```

Perceba que existe um espaço entre os três pontos `...` e o nome do parâmetro. Apesar de estar incorreto pela PSR-12, o código funcionará perfeitamente.

Regra 35

Ao combinar o operador de referência `&` e o operador variável de três pontos `...`, NÃO DEVE haver nenhum espaço entre os dois. Essa regra é bem semelhante à anterior, vamos analisar o exemplo a seguir para vermos onde está a diferença:

```
<?php

declare(strict_types=1);

namespace App\Crypt;

class Crypt
{
    public function crypt(string $string, &...$parts): string
    {
        //Código
    }
}
```

A única diferença apresentada nesse exemplo é a combinação do operador de referência `&` e o operador variável de três pontos, `...`. As demais regras de espaçamento se aplicam identicamente a essa regra.

E como seria um dos modos considerado incorreto pela PSR-12? Seria da seguinte maneira:

```
<?php

declare(strict_types=1);

namespace App\Crypt;
```

```
class Crypt
{
    public function crypt(string $string, & ... $parts): string
    {
        //Código
    }
}
```

Conclusão

Como você pôde ver são inúmeras regras que a PSR-12 disponibiliza, e ainda tem mais! No próximo capítulo continuaremos vendo quais são as demais regras da PSR-12. Então, siga em frente!

Capítulo 11

PSR-12 (EXTENDED CODING STYLE) - PARTE II

Vimos no capítulo anterior um total de 35 regras da PSR-12. Neste capítulo vamos continuar vendo essas regras que visam facilitar o estilo do nosso código-fonte para obtermos uma melhor leitura e compreensão através dos padrões estabelecidos pela PSR-12. Lembrando que a PSR12 é uma extensão da PSR-2 e por esse motivo a PSR-2 foi descontinuada.

11.1 REGRAS

A seguir vamos continuar com as regras de onde paramos no capítulo anterior.

Regra 36

As palavras-chaves `abstract` e `final` DEVEM ser declaradas antes de definir a visibilidade de classes e métodos. Há casos em que precisamos criar uma classe abstrata ou final, por exemplo, ou até mesmo métodos, e existe uma maneira correta de fazer isso.

Para simplificar o entendimento, vamos analisar o exemplo a

seguir, que demonstra a maneira correta de seguir essa regra:

```
<?php

declare(strict_types=1);

namespace App\Sercice;

abstract class AbstractService
{
    protected static $dados;

    abstract protected function insert(): bool;

    final public static function start(): bool
    {
        //Código
    }
}
```

Perceba que as palavras-chaves `abstract` e `final` precedem a visibilidade dos dois métodos `insert()` e `start()`. Na classe não existe a visibilidade para ser definida, mas podemos informar se é uma classe abstrata ou final.

Perceba também que a palavra-chave `static` está definida após a visibilidade do método e da propriedade.

É um exemplo simples, mas que demonstra a utilização dessas palavras-chaves. Agora veremos um exemplo que demonstra a maneira incorreta de utilizá-las:

```
<?php

namespace App\Sercice;

class abstract AbstractService
{
    protected $dados;

    protected abstract function insert(): bool;
```

```
    public final static function start(): bool
    {
        //Código
    }
}
```

Nesse exemplo, as palavras-chaves `abstract` e `final` encontram-se logo após a visibilidade dos métodos e também logo após a palavra-chave `class` .

Esse exemplo já não funcionará pois não podemos definir uma classe abstrata após a palavra-chave `class` , sendo permitido apenas antes.

Perceba que nos métodos também existem as palavras `abstract` e `final` após a definição da visibilidade, e não há problemas. Porém, se fizer deste modo, seu código não ficará de acordo com a regra estabelecida.

Regra 37

Ao declarar um método ou função NÃO DEVE existir espaço entre o nome do método/função e o parêntese de abertura.

NÃO DEVE existir espaço após o parêntese de abertura e NÃO DEVE existir espaço antes do parêntese de fechamento; e na lista de parâmetros NÃO DEVE existir espaço antes de cada vírgula, mas DEVE existir um espaço após cada vírgula. Parece complexo, não é mesmo? Mas não é complicada, na verdade é tão simples quanto definir uma variável. Veja o exemplo a seguir que demonstra a maneira correta da aplicação dessa regra:

```
<?php

declare(strict_types=1);
```

```php
namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados): bool
    {
        //Código
    }
}
```

Simples, não? Note que quando definimos o nome do método não existe espaço antes e após o parêntese de abertura, não existe espaço antes da vírgula entre os parâmetros e não existe espaço antes e após o parêntese de fechamento.

Agora vamos analisar um exemplo considerando a maneira incorreta e que não estará em conformidade com essa regra da PSR:

```php
<?php

declare(strict_types=1);

namespace App\Sercice;

class AbstractService
{
    protected function update ( $id , array $dados ): bool
    {
        //Código
    }
}
```

Esse exemplo está completamente fora da regra, pois existe espaço em tudo quanto é lugar na definição do método, evite ao máximo utilizar esse estilo de código.

Regra 38

A lista de parâmetros pode ser dividida em várias linhas, onde cada linha subsequente é recuada uma vez. Ao fazer isso, o primeiro item da lista DEVE estar na próxima linha e DEVE haver apenas um parâmetro por linha.

Um único parâmetro dividido em várias linhas (como pode ser o caso de uma função ou matriz anônima) não constitui a divisão da própria lista de parâmetros.

Vamos analisar o exemplo a seguir para tornar mais clara a utilização dessa regra:

```
<?php

$usuario->setDados(
    $nome,
    $sobrenome,
    $idade
)
```

Perceba que assim como podemos quebrar a definição da lista de parâmetros dos métodos e funções, também podemos quebrar em linhas a lista de parâmetros passados para o método/função.

Vamos analisar um outro exemplo que segue a mesma ideologia do exemplo anterior, porém com outros exemplos que são permitidos pela PSR-12, inclusive utilizando função anônima:

```
<?php

$usuario->setDados($nome, $sobrenome, [
    'idade' => 29,
    'data_nascimento' => '1990-06-06'
], $sexo);

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello ' . $app->escape($name);
});
```

Regra 39

Após a palavra-chave da estrutura de controle DEVE existir um espaço. NÃO DEVE existir um espaço após o parêntese de abertura (se necessário).

NÃO DEVE existir espaço antes do parêntese de fechamento (se necessário). DEVE existir um espaço entre o parêntese de fechamento e a chave de abertura.

O corpo da estrutura de controle DEVE ser indentado uma vez. A chave de fechamento DEVE ser colocada uma linha após o corpo da estrutura de controle. Essa regra também parece assombrosa, mas ela é simples. Volto a dizer que é mais uma questão de adaptação e costume do que uma dificuldade lógica propriamente dita.

Vamos analisar um exemplo para cada estrutura de controle, que demonstra a maneira correta a ser seguida:

Exemplos com `if`, `elseif`, `else`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        if (is_int($id)) {
            $resultado = 'ID é um inteiro';
        } elseif (is_numeric($id)) {
            $resultado = 'ID é uma string contendo apenas números
;
        } else {
```

```
            $resultado = 'ID não possui um formato válido';
        }
    }
}
```

Perceba a indentação da estrutura de controle, o espaço antes do parêntese de abertura e ainda o espaço após o parêntese de fechamento antes da chave de abertura da estrutura de controle.

Expressões entre parênteses podem ser divididas em várias linhas, sendo que cada linha subsequente é recuada pelo menos uma vez. Ao fazer isso, a primeira condição DEVE estar na próxima linha. O parêntese de fechamento `)` e a chave de abertura `{` DEVEM ser colocados juntos em sua própria linha, com um espaço entre eles. Operadores booleanos entre condições DEVEM estar sempre no início ou no final da linha, não uma mistura de ambos.

Vamos analisar um exemplo de aplicação dessa regra:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        if (
            is_string($id)
            && is_numeric($id)
        ) {
            $resultado = 'ID é um número';
        } elseif (
            is_numeric($id)
            && is_int($id)
        ) {
```

```
            $resultado = 'ID é uma string contendo apenas números
;
        } else {
            $resultado = 'ID não possui um formato válido';
        }
    }
}
```

Não é uma regra complicada a ser seguida, mas deve-se tomar cuidado para não definir incorretamente a estrutura de controle e é isso que veremos no exemplo a seguir, que demonstra a maneira considerada incorreta pela PSR-12 e que deve ser evitada:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        if(is_int($id)){
            $resultado = 'ID é um inteiro';
        }
        elseif(is_numeric($id)){
            $resultado = 'ID é uma string contendo apenas números
;
        }
        else{
            $resultado = 'ID não possui um formato válido';
        }
    }
}
```

Já nesse exemplo, apesar de funcionar corretamente, ele não segue a regra da PSR-12. Como você pode ver, não existe espaço antes do parêntese de abertura `(` e nem após o parêntese de fechamento `)`.

Outra regra quebrada é quanto às palavras-chaves `elseif` e `else` que estão abaixo da chave de fechamento `}` da estrutura anterior. O correto é estarem na mesma linha que a chave de fechamento `}` e precedidas de um único espaço.

Exemplos com `switch`, `case`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        switch ($id) {
            case 0:
                $resultado = 'Id é 0';
                break;
            case 1:
                $resultado = 'Id é 1';
                break;
            case 2:
                $resultado = 'Id é 2';
                break;
            case 3:
                $resultado = 'Id é 3';
                //Sem break
            default:
                $resultado = 'O valor informado não é permitido';
                break;
        }
    }
}
```

Perceba como é simples: após a palavra-chave `switch` DEVE existir um espaço antes do parêntese de abertura `(` e DEVE existir um espaço após o parêntese de fechamento `)`.

Cada `case` DEVE ser indentado uma única vez e o conteúdo de cada `case` também DEVE ser indentado uma única vez. A palavra-chave `break` DEVE estar indentada no mesmo nível do conteúdo do `case`.

Quando não houver `break` DEVE existir um comentário `//no break` no caso.

Esse exemplo está perfeitamente em conformidade com a PSR-12 e deve ser utilizado dessa maneira.

Vamos analisar um exemplo de aplicação da mesma regra de formatação da lista de expressões em múltiplas linhas quando utilizamos o `switch`:

```
<?php

declare(strict_types=1);

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados): string
    {
        switch (
            is_int($id)
            && !empty($dados)
        ) {
            case true:
                $resultado = 'Update recebeu valores válidos';
                break;
            default:
                $resultado = 'Update não recebeu valores válidos'
;
                break;
        }
    }
}
```

Agora vamos analisar um exemplo que é considerado incorreto quanto ao estilo aplicado na regra:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $resultado = null;

        switch($id){
            case 0:
                $resultado = 'Id é 0';
            break;
            case 1:
                $resultado = 'Id é 1';
            break;
            case 2:
                $resultado = 'Id é 2';
            break;
            case 3:
                $resultado = 'Id é 3';
            default:
                $resultado = 'O valor informado não é permitido';
            break;
        }
    }
}
```

Perceba que nesse exemplo a estrutura sofreu alterações de indentação e espaçamento, e também que no `case` que não possui o `break` o comentário `//no break` foi removido.

O método continua funcionando perfeitamente, porém não é válido para a PSR-12.

Exemplos com `while` , `do while`

```
<?php
```

```
namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        while ($id < 10) {
            //Código
        }

        do {
            //Código
        } while ($id > 0)
    }
}
```

Após a palavra-chave `while`, DEVE existir um espaço antes do parêntese de abertura `(` e DEVE existir um espaço após o parêntese de fechamento `)`.

A abertura da chave `{` deve estar na mesma linha da declaração `while`. O mesmo se aplica utilizando o `do while`.

Vamos analisar um exemplo de aplicação da mesma regra de formatação da lista de expressões em múltiplas linhas quando utilizamos os laços `while` e `do while`:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        while (
            $id < 10
            && !empty($dados)
        ) {
            //Código
```

```
        }

        do {
            //Código
        } while (
            $id > 0
            && !empty($dados)
        )
    }
}
```

O código fica mais bonito e legível, e essa é a maneira considerada correta pela PSR-12. Agora vamos ver o exemplo a seguir, que demonstra a maneira incorreta e que não deve ser seguida:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        while(
            $id < 10
            && !empty($dados)
        ){
            //Código
        }

        do{
            //Código
        }while(
            $id < 10
            && !empty($dados)
        )
    }
}
```

O exemplo é semelhante ao da maneira correta, a única diferença aqui são os espaçamentos que não existem entre as

palavras-chaves e o parêntese de abertura `(` e entre o parêntese de fechamento `)` e a chave de abertura `{`.

O exemplo funciona normalmente, mas não é válido para a PSR-12.

Exemplos com `for`

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        for ($i = 0; $i <= 10; $i++) {
            //Código
        }
    }
}
```

Nesse exemplo foi aplicado exatamente como no exemplo da utilização correta com `while`, ou seja, os espaçamentos estão de acordo com a regra da PSR-12 e é considerado a maneira correta.

Perceba novamente que, após a palavra-chave `for`, DEVE existir um espaço antes do parêntese de abertura `(` e DEVE existir um espaço após o parêntese de fechamento `)`.

Vamos analisar um exemplo de aplicação da mesma regra de formatação da lista de expressões em múltiplas linhas quando utilizamos o laço `for`:

```
<?php

namespace App\Sercice;

class AbstractService
```

```
{
    protected function update($id, array $dados)
    {
        for (
            $i = 0;
            $i <= 10;
            $i++
        ) {
            //Código
        }
    }
}
```

Vamos analisar agora o exemplo considerado incorreto e que não é aceito pela PSR-12:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        for($i = 0;$i <= 10;$i++){
            //Código
        }
    }
}
```

Nesse exemplo, o código também funciona corretamente, porém não é aceito pela PSR-12 pois não possui os espaçamentos definidos.

Exemplo com `foreach`

```
<?php

namespace App\Sercice;

class AbstractService
{
```

```
    protected function update($id, array $dados)
    {
        foreach ($dados as $key => $value) {
            //Código
        }
    }
}
```

Esse exemplo é semelhante ao aplicado com o `for` e as mesmas regras de espaçamento também se aplicam utilizando o `foreach`. Esse exemplo possui os espaçamentos que são considerados corretos pela PSR-12 e deve ser utilizado.

Vamos analisar agora o mesmo exemplo, porém com a ausência dos espaçamentos e que é considerado incorreto:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        foreach($dados as $key=>$value){
            //Código
        }
    }
}
```

Nada demais até aqui, o exemplo funciona normalmente, mas é considerado incorreto pela PSR-12 devido à ausência de espaços em sua definição.

Exemplos com `try`, `catch`, `finally`

```
<?php

namespace App\Sercice;

class AbstractService
```

```
{
    protected function update($id, array $dados)
    {
        try {
            //Código
        } catch (\InvalidArgumentException $e) {
            //Código
        } catch (\ArithmeticError | \Exception $e) {
            //Código
        } finally {
            //Código
        }
    }
}
```

Quando houver a necessidade de utilizar o bloco `try catch` ou `try catch finally`, ele deverá ter a estrutura de espaçamentos e indentação mostrados no exemplo anterior.

Após a palavra-chave `try` DEVE existir um espaço antes da abertura da chave `{` e DEVE existir um espaço após a chave de fechamento `}`.

Para cada bloco `catch` DEVE existir um espaço após a palavra-chave `catch` e DEVE existir um espaço após o parêntese de fechamento `)`.

O mesmo se aplica ao `finally` (quando existir), após a palavra-chave `finally`, DEVE existir um espaço antes da chave de abertura `{` do bloco.

Esse exemplo está completamente em conformidade com a regra da PSR-12. Vamos analisar um exemplo em que essa regra não é seguida e é considerado incorreto:

```
<?php

namespace App\Sercice;
```

```
class AbstractService
{
    protected function update($id, array $dados)
    {
        try{
            //Código
        }
        catch(\InvalidArgumentException $e){
            //Código
        }
        catch(\ArithmeticError|Exception $e){
            //Código
        }
        finally{
            //Código
        }
    }
}
```

No exemplo anterior todo o código foi alterado e não há espaçamento onde deveria ter e a indentação também está incorreta, não sendo aceito pela PSR-12.

Regra 40

As regras de estilo para os operadores são agrupadas pelo número de operandos que eles usam).

Quando o espaço é permitido em torno de um operador, vários espaços PODEM ser usados para fins de legibilidade.

Operadores Unários

Os operadores de incremento/decremento NÃO DEVEM ter espaços entre o operador e o operando. O exemplo a seguir demonstra a aplicação dessa regra:

```
<?php
```

```
$i++;
++$j;

$i--;
--$i
```

Essa regra diz que entre o `$i` e o operador `++` não devemos ter espaços. O mesmo vale para as demais variações de ambas as variáveis.

O exemplo a seguir demonstra o modo considerado incorreto pela PSR-12:

```
<?php

$i ++;
++ $j;

$i --;
-- $i
```

Os operadores de conversão de tipo NÃO DEVEM ter nenhum espaço entre parênteses:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $id = (int) $id;
    }
}
```

Perceba que, ao realizar o `casting` do `$id` para inteiro `int` entre parênteses, `(int)`, não existe nenhum espaçamento. Esse é o modo considerado correto pela PSR-12 e é a maneira que a

maioria dos desenvolvedores utiliza.

A maneira considerada incorreta para essa regra seria contendo os espaços dentro dos parênteses do `casting` :

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $id = ( int ) $id;
    }
}
```

Operadores Binários

Todos os operadores de aritmética binária, comparação, atribuição, bit a bit, lógico, string e tipo DEVEM ser precedidos e seguidos por pelo menos um espaço. O exemplo a seguir demonstra a aplicação dessa regra e muitos desenvolvedores também já estão acostumados a ela:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        if ($id === 1) {
            $id = $dados ?? $id ?? 2;
        } elseif ($id > 1) {
            $id = (1 + 1) * $id;
        }
    }
}
```

Perceba que antes e após a definição de cada operador existem um espaço e é exatamente isso que torna essa regra válida para a PSR-12.

O exemplo a seguir demonstra a maneira considerada incorreta, ou seja, sem os espaçamentos antes e depois dos operadores:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        if ($id===1) {
            $id = $dados??$id??2;
        } elseif ($id>1) {
            $id = (1+1)*$id;
        }
    }
}
```

Perceba que o código ficou bem ilegível em alguns trechos, e foi por isso que a PSR-12 surgiu.

Operadores Ternários

O operador condicional, também conhecido simplesmente como operador ternário, DEVE ser precedido e seguido por pelo menos um espaço em torno de ambos caracteres `?` e `:`. Vamos analisar o exemplo a seguir, que demonstra a aplicação dessa regra:

```
<?php

namespace App\Sercice;

class AbstractService
```

```
{
    protected function update($id, array $dados)
    {
        $id = $id ? $id : 1;
    }
}
```

No exemplo, perceba os espaços antes e após a definição dos caracteres `?` e `:`. Agora vamos ver como seria o modo considerado incorreto pela PSR-12:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $id = $id?$id:1;
    }
}
```

Perceba como a remoção dos espaços torna o código mais ilegível, impossibilitando identificar a condição de maneira fácil e rápida.

Quando o operando do meio do operador condicional é omitido, o operador DEVE seguir as mesmas regras de estilo que outros operadores de comparação binária. Vamos ver o exemplo a seguir para melhor entendimento:

```
<?php

namespace App\Sercice;

class AbstractService
{
    protected function update($id, array $dados)
    {
        $id = $id ?: 1;
```

```
    }
}
```

Regra 41

Assim como nos métodos, a lista de parâmetros da `closure` (função anônima) PODE ser dividida em múltiplas linhas, sendo que cada linha DEVE ser indentada uma única vez.

Sempre que isso ocorrer, o primeiro parâmetro DEVE estar na próxima linha e DEVE existir apenas um parâmetro por linha.

Quando a lista de parâmetros for dividida em múltiplas linhas, o parêntese de fechamento `)` e a chave de abertura `{` DEVEM estar na mesma linha, contendo apenas um espaço entre ambos.

Difícil? Não se preocupe, já vimos o funcionamento quando aplicamos essa regra utilizando métodos. Vamos ao exemplo considerado correto pela PSR-12 e que deve ser seguido:

```php
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function (
            $atualizado,
            $usuario
        ) use (
            $id,
            $dados
        ) {
            //Código
        }
    }
}
```

Veja como o código é dividido e indentado seguindo corretamente a regra estabelecida pela PSR-12. Vamos analisar um exemplo considerado incorreto:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
    public function updateAction($id, array $dados)
    {
        return function ($atualizado,
            $usuario) use ($id,
            $dados) {
            //Código
        }
    }
}
```

O exemplo anterior demonstra exatamente o que não fazer. Não indente o código dessa maneira, por mais que funcione. Se você fizer isso, seu código não estará em conformidade com a PSR-12.

Regra 42

Se um tipo de retorno estiver presente, DEVE seguir as mesmas regras que as funções e métodos normais; se a palavra-chave `use` estiver presente, os dois pontos `:` DEVE seguir a lista de uso entre parênteses sem espaços entre os dois caracteres. O exemplo a seguir demonstra como aplicar essa regra corretamente:

```
<?php

namespace App\Controller\Usuario;

class UsuarioController
{
```

```
    public function updateAction($id, array $dados)
    {
        $closure = function (
            $atualizado,
            $usuario
        ) use (
            $id,
            $dados
        ): bool {
            //Código
        }
    }
}
```

Perceba o tipo de retorno definido na `closure`, no caso `bool`. Existe um espaço após os dois pontos `:` e antes da chave de abertura `{` da `closure`.

Regra 43

Listas de parâmetros e listas de variáveis PODEM ser divididas em várias linhas, sendo que cada linha subsequente é recuada uma vez. Ao fazer isso, o primeiro item da lista DEVE estar na próxima linha e DEVE haver apenas um parâmetro ou variável por linha.

Quando a lista final (seja de parâmetros ou variáveis) é dividida em várias linhas, o parêntese de fechamento `)` e a chave de abertura `{` DEVEM ser colocados juntos em sua própria linha, com um espaço entre eles.

A seguir vamos analisar exemplos que demonstram a aplicação dessa regra:

```
<?php

$closureSemVariaveis = function (
    $param1,
    $param2,
```

```
    $param3
) {
   // Código
};

$closureSemParametroEComVariaveis = function () use (
    $var1,
    $var2,
    $var3
) {
   // Código
};

$closureComParametrosEVariaveis = function (
    $param1,
    $param2,
    $param3
) use (
    $var1,
    $var2,
    $var3
) {
   // Código
};

$closureComParametrosEUmaVariavel = function (
    $param1,
    $param2,
    $param3
) use ($var1) {
   // Código
};

$closureComUmParametroEVariasVariaveis = function ($param1) use (
    $var1,
    $var2,
    $var3
) {
   // Código
};
```

Perceba como é realizada a indentação e os espaçamento para cada um dos casos. A seguir vamos ver um exemplo em que as

regras de formatação também se aplicam quando o fechamento é usado diretamente em uma chamada de função ou como parâmetro de um método:

```
<?php

$object->method(
    $param1,
    function ($param2) use ($var1) {
        // Código
    },
    $param3
);
```

Regra 44

Classes anônimas DEVEM seguir as mesmas diretrizes e princípios que os fechamentos que vimos anteriormente. Vamos ver um exemplo de como definir uma classe anônima compatível com a PSR-12:

```
<?php

$instance = new class {};
```

Regra 45

A chave de abertura `{` PODE estar na mesma linha que a palavra-chave da classe, desde que a lista de interfaces não seja finalizada. Se a lista de interfaces terminar, a chave DEVE ser colocada na linha imediatamente após a última interface.

Vamos conhecer como essa regra é aplicada na prática através do exemplo a seguir:

```
<?php
```

```php
//Chave de abertura na mesma linha
$instance = new class extends AbstractClass implements \Handleabl
eInterface {
    //Código
};

//Chave de abertura na próxima linha
$instance = new class extends AbstractClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    //Código
};
```

Perceba que o espaçamento e a indentação seguem princípios que já vimos anteriormente nessa PSR.

Conclusão

Chegamos ao final da PSR-12, como pôde ser visto nessa PSR, há várias regras e condições que devem ser seguidas para que seu código esteja em conformidade. Aplicamos regras de espaços e indentação que claramente mostraram que o código fica bem mais fácil de ser entendido.

No primeiro momento pode parecer estranho como o código fica, mas com o tempo você acaba se acostumando e até mesmo decorando como cada regra é aplicada. Vamos seguir em frente e conhecer a PSR-13 e suas regras.

CAPÍTULO 12

PSR-13 (LINK DEFINITION INTERFACES)

Essa PSR especifica uma maneira simples e comum de representar link de hipermídia independentemente do formato de serialização utilizado, permitindo que a aplicação serialize uma resposta com links de hipermídia em um ou mais formatos.

12.1 REGRAS

Nesta seção vamos conhecer as regras que definem a PSR-13, mas antes precisamos conhecer algumas características:

Links básicos - Um link de hipermídia consiste em: um `URI` que representa o recurso de destino que está sendo referenciado e um relacionamento que define como o recurso de destino se relaciona com a origem. Vários outros atributos do link podem existir dependendo do formato utilizado.

- **Instanciando um objeto** - Um objeto deve ser criado através da implementação de uma das interfaces definidas nesta PSR.
- **Serializador** - Uma aplicação que utiliza um ou mais

objetos `Link` e produz uma representação serializada em algum formato definido.

Regra 1

Todos os links PODEM incluir zero ou mais atributos adicionais além do `URI` e do relacionamento. Não há uma descrição dos valores permitidos e a validade dos valores depende do contexto e, geralmente, de um formato de serialização específico. Os valores comumente suportados são `hreflang`, `title` e `type`.

Regra 2

Serializadores PODEM omitir atributos em um objeto de link caso haja a necessidade, porém, os serializadores DEVEM codificar todos os atributos fornecidos, a fim de permitir a extensão do usuário a menos que seja impedido pela definição de algum formato de serialização.

Alguns atributos (geralmente o `hreflang`) podem aparecer mais de uma vez em seu contexto. Portanto, o atributo pode possuir valores em forma de `array` (matriz) em vez de valores simples.

Os serializadores PODEM codificar esse `array` em qualquer formato apropriado para o formato serializado.

Regra 3

Caso um determinado atributo não possa ter múltiplos valores em seu contexto específico, os serializadores DEVEM utilizar o

primeiro valor fornecido e ignorar todos os valores subsequentes. Se o atributo possuir um valor booleano `true`, os serializadores PODEM usar formas abreviadas, se apropriadas e suportadas por algum formato de serialização. Essa regra se aplica se, e somente se, o atributo for booleano `true`, ou seja, o valor 1 utilizado como `true` não é válido.

Regra 4

Caso o atributo possua um valor booleano `false`, os serializadores DEVEM omitir o atributo, a menos que o significado semântico do resultado seja alterado. Esta regra se aplica se, e somente se, o atributo for booleano `false`, ou seja, o valor 0 utilizado para representar `false` não é válido.

Regra 5

Relacionamentos de links são definidos como string e são simplesmente uma palavra-chave no caso de uma relação definida publicamente ou um `URI` absoluto no caso de relações privadas.

Caso uma palavra-chave simples seja utilizada, DEVE corresponder a um registro do `IANA` (`http://www.iana.org/assignments/link-relations/link-relations.xhtml`). Opcionalmente, o registro contido em `http://microformats.org/wiki/existing-rel-values` PODE ser utilizado, mas pode não ser válido em todos os contextos.

Regra 6

Um relacionamento que não esteja definido em um dos

registros citados anteriormente ou em um registro público é considerado privado, isto é, é específico de uma determinada aplicação. Essas relações DEVEM utilizar um `URI` absoluto.

Regra 7

Modelos de link são um padrão para um `URI` que deve ser preenchido com valores enviados por um cliente. Alguns formatos de hipermídias suportam modelos de links, enquanto outros não, e podem ter uma maneira diferente de denotar que um link é um modelo.

Um serializador que não suporta modelos de `URI` DEVE ignorar quaisquer links modelados que encontrar.

Regra 8

Provedores evoluídos - Um provedor de links poderá precisar da capacidade de ter links adicionais adicionados a ele. Em outros casos, um provedor de link é somente leitura, com links derivados em tempo de execução de outra fonte de dados, por esse motivo, provedores de links modificáveis são uma interface secundária que opcionalmente pode ser implementada.

Além disso, alguns objetos provedores de link como objetos de resposta da PSR-7 teriam os métodos para adicionar links a eles totalmente incompatíveis por possuírem um design imutável. Portanto, o método definido pela interface `Psr\Link\EvolvableLinkProviderInterface` exige que um novo objeto idêntico ao original seja devolvido, mas contendo um objeto de link adicional incluído.

Objetos de link evoluídos - A PSR-13 permite que os objetos possam evoluir da mesma maneira que os objetos da PSR-7. Por essa razão, um objeto `Psr\Link\EvolvableLinkInterface` adicional está incluído e disponibiliza métodos para produzir novas instâncias de objeto com uma única alteração.

Graças ao comportamento `copy-on-write` do PHP, ainda é eficiente em termos de `CPU` e memória. Não existe um método evolutivo para modelos, porém, como o valor é um modelo de um link, então é baseado exclusivamente no valor de `href` .

Um objeto de link evoluído NÃO DEVE ser definido de forma independente.

12.2 INTERFACES

Agora que conhecemos as regras desta PSR, vamos conhecer a seguir, as interfaces que fazem parte deste pacote.

`Psr\Link\LinkInterface`

```
<?php

namespace Psr\Link;

interface LinkInterface
{
    public function getHref();
    public function isTemplated();
    public function getRels();
    public function getAttributes();
}
```

`Psr\Link\EvolvableLinkInterface`

```
<?php
```

```
namespace Psr\Link;

interface EvolvableLinkInterface extends LinkInterface
{
    public function withHref($href);
    public function withRel($rel);
    public function withoutRel($rel);
    public function withAttribute($attribute, $value);
    public function withoutAttribute($attribute);
}
```

Psr\Link\LinkProviderInterface

```
<?php

namespace Psr\Link;

interface LinkProviderInterface
{
    public function getLinks();
    public function getLinksByRel($rel);
}
```

Psr\Link\EvolvableLinkProviderInterface

```
<?php

namespace Psr\Link;

interface EvolvableLinkProviderInterface extends LinkProviderInte
rface
{
    public function withLink(LinkInterface $link);
    public function withoutLink(LinkInterface $link);
}
```

Conclusão

Chegamos ao final da PSR-13, que define um conjunto de interfaces para a definição de links. A PSR em si não é grande e pode parecer um pouco confusa, mas na prática, quando os

conceitos são aplicados, ela não é tão complexa.

No próximo capítulo vamos abordar sobre a PSR-14, que trata conceitos de *Event Dispatcher*, permitindo que os desenvolvedores insiram lógica em uma aplicação de maneira fácil e consistente. Então vamos nessa!

Capítulo 13

PSR-14 (EVENT DISPATCHER)

O principal objetivo dessa PSR é estabelecer um mecanismo comum para a extensão e colaboração baseada em eventos, de modo que bibliotecas e componentes possam ser reutilizados mais livremente entre várias aplicações.

Ter interfaces comuns para despachar e manipular eventos permite aos desenvolvedores a liberdade de criarem bibliotecas que podem interagir com muitas estruturas e outras bibliotecas.

Podemos citar alguns exemplos de uso, a seguir:

- Uma estrutura de segurança que impedirá a gravação/acesso a dados quando um usuário não possuir permissão.
- Um sistema comum de cache de página inteira.
- Bibliotecas que estendem outras bibliotecas, independentemente de quais estruturas estão integradas.
- Rastrear todas as ações executadas na aplicação.

Conhecendo um pouco mais sobre os elementos do Event Dispatcher, nós temos:

- **`Event`** (Evento) - É uma mensagem produzida por um emissor. Pode ser qualquer objeto PHP arbitrário.
- **`Listener`** (Ouvinte) - É qualquer chamada do PHP que espera receber um evento. Zero ou mais ouvintes podem ser passados no mesmo evento. Um ouvinte PODE enfileirar algum outro comportamento assíncrono, se assim desejar.
- **`Emitter`** (Emissor) - É qualquer código arbitrário que deseje despachar um evento. Isso também é conhecido como o "código de chamada". Ele não é representado por nenhuma estrutura de dados específica, mas se refere ao caso de uso.
- **`Dispatcher`** (Despachante) - É um objeto de serviço que recebe um objeto `Event` por um `Emitter`. O Despachante é responsável por garantir que o evento seja passado a todos os ouvintes relevantes, mas DEVE adiar a determinação dos ouvintes responsáveis para um provedor de escuta.
- **`Listener Provider`** (Provedor de Ouvinte) - É responsável por determinar quais ouvintes são relevantes para um determinado evento, mas NÃO DEVE chamar os ouvintes em si. Um `Listener Provider` pode especificar zero ou mais ouvintes relevantes.

13.1 REGRAS

Vamos conhecer a seguir as regras que definem essa PSR.

Regra 1

Eventos são objetos que atuam como a unidade de

comunicação entre um `Emitter` (Emissor) e `Listener` (Ouvinte) apropriado.

Os objetos `Event` PODEM ser mutáveis, ou seja, quando o caso de uso chama os `Listeners` fornecendo informações de volta ao `Emitter` (Emissor). Porém, se nenhuma comunicação bidirecional for necessária, é RECOMENDADO que o evento seja definido como imutável, isto é, definido de tal modo que não tenha métodos de mutação.

Regra 2

As aplicações DEVEM assumir que o mesmo objeto será passado para todos os `Listeners` .

Regra 3

É RECOMENDADO, MAS NÃO NECESSÁRIO, que os objetos `Event` suportem serialização e desserialização sem perdas, ou seja, `$event == unserialize(serialize($event))` DEVE ser verdadeiro.

Regra 4

Objetos PODEM alavancar a interface serializável do PHP, os métodos mágicos `__sleep()` ou `__wakeup()` ou a funcionalidade semelhante.

Regra 5

Um evento `Stoppable` é um caso especial de evento que contém formas adicionais de impedir que outros `Listeners`

sejam chamados. É indicado implementando a interface `StoppableEventInterface`.

Um evento que implementa a interface `StoppableEventInterface` DEVE retornar `true` do método `isPropagationStopped()` quando o evento que ele representa for concluído. É de responsabilidade da aplicação determinar quando o evento deve ou não ser concluído.

Por exemplo, um evento que esteja solicitando que um objeto PSR-7 `RequestInterface` seja correspondido com um objeto `ResponseInterface` correspondente poderia ter um método `setResponse(ResponseInterface $response)` para um `Listener` chamar, o que faria com que o método `isPropagationStopped()` retornasse `true`.

Regra 6

Um `Listener` PODE ser chamado por qualquer código PHP. Um `Listener` DEVE ter apenas um parâmetro, que é o evento ao qual ele responde. Os `Listeners` DEVEM indicar a dica desse parâmetro de acordo com a relevância do caso de uso. Uma dica do tipo de `Listener` PODE indicar para uma interface que é compatível com qualquer tipo de evento que realiza uma implementação específica ou não dessa interface.

Regra 7

Um `Listener` DEVE ter um retorno nulo e DEVE informar a dica que retorna explicitamente. Um `Dispatcher` deve ignorar os valores de retorno dos `Listeners` (Ouvintes).

Regra 8

Um `Listener` PODE delegar ações para outro código. Isso inclui um `Listener` sendo um `wrapper` (embrulho) em torno de um objeto que executa a lógica de negócios real.

Regra 9

Um `Listener` PODE enfileirar informações do evento para processamento posterior por um processo secundário, usando o cron, um servidor de filas ou técnicas semelhantes.

Um `Listener` pode serializar o próprio objeto `Event` para fazer isso. Um processo secundário DEVE presumir que quaisquer alterações feitas em um objeto `Event` NÃO serão propagadas para outros `Listeners` .

Regra 10

Um `Dispatcher` é um objeto de serviço que implementa a interface `EventDispatcherInterface` . Ele é responsável por recuperar os `Listeners` de um `Listener Provider` para o evento despachado e invocar cada `Listener` com esse evento.

Um `Dispatcher` :

- DEVE chamar os `Listeners` sincronicamente na ordem em que eles são retornados de um `Listener Provider` .
- DEVE retornar o mesmo objeto `Event` que foi passado depois de ser feito, chamando `Listeners` .
- NÃO DEVE retornar ao emissor até que todos os `Listeners` tenham executado.

Regra 11

Se passou um evento `Stoppable`, um `Dispatcher` DEVE chamar o método `isPropagationStopped()` no evento antes que cada `Listener` seja chamado. Se esse método retornar `true`, ele DEVE retornar o evento ao `Emitter` (Emissor) imediatamente e NÃO DEVE chamar mais nenhum `Listener`.

Se um evento for passado para o `Dispatcher` que sempre retorna `true` através do método `isPropagationStopped()`, zero `Listeners` será chamado.

Regra 12

Um `Dispatcher` deve assumir que qualquer `Listener` retornado a ele de um `ListenerProvider` é seguro para o tipo, ou seja, o `Dispatcher` DEVE assumir que chamar `$listener($event)` não produzirá um `TypeError`.

Regra 13

Uma exceção ou erro lançado por um `Listener` DEVE bloquear a execução de outros `Listeners` e DEVE ter permissão para se propagar de volta para o emissor.

Regra 14

Um `Dispatcher` PODE pegar um objeto para registrá-lo, permitir que medidas adicionais sejam tomadas etc., mas, em seguida, DEVE recriar o item original.

Regra 15

Um `Listener Provider` é um objeto de serviço responsável por determinar quais `Listeners` são relevantes e devem ser chamados para um determinado evento. Pode determinar tanto que os `Listeners` são relevantes quanto a ordem pela qual retorná-los.

Isso PODE incluir:

- Permitir alguma forma de mecanismo de registro para que as aplicações possam atribuir um `Listener` a um evento em uma ordem fixa.
- Derivar uma lista de `Listeners` aplicáveis por meio da reflexão com base no tipo e nas interfaces implementadas do evento.
- Gerar uma lista compilada de `Listeners` antecipadamente que podem ser consultados em tempo de execução.
- Implementar alguma forma de controle de acesso para que determinados `Listeners` sejam chamados apenas se o usuário atual tiver uma determinada permissão.
- Extrair algumas informações de um objeto referenciado pelo evento, como uma entidade, e chamando métodos de ciclo de vida predefinidos nesse objeto.
- Delegar sua responsabilidade a um ou mais `Listener Proveiders` usando alguma lógica arbitrária.

Qualquer combinação dos itens citados anteriormente PODE ser usada como desejado.

Regra 16

`Listener Providers` DEVEM usar o nome da classe de um evento para diferenciar um evento do outro. Eles também PODEM considerar qualquer outra informação sobre o evento, conforme seja mais apropriado.

Regra 17

Os `Listener Providers` DEVEM tratar os tipos de pai de forma idêntica ao tipo do próprio evento ao determinar a aplicabilidade do `Listener` . No seguinte caso:

```
<?php
class A {}

class B extends A {}

$b = new B();

function listener(A $event): void {};
```

Um `Listener Provider` DEVE tratar o método `listener()` como um `Listener` aplicável por `$b` , pois é compatível com o tipo, a menos que outros critérios o impeçam de fazê-lo.

Regra 18

Um `Dispatcher` DEVE compor um `Listener Provider` para determinar os `Listeners` relevantes. É RECOMENDADO que um `Listener Provider` seja implementado como um objeto distinto do `Dispatcher` , mas NÃO É NECESSÁRIO.

13.2 INTERFACES

A seguir vamos conhecer as interfaces que fazem parte desse pacote.

`Psr\EventDispatcher\EventDispatcherInterface`

Esta interface define um `Dispatcher` para eventos.

```
<?php
namespace Psr\EventDispatcher;

interface EventDispatcherInterface
{
    public function dispatch(object $event);
}
```

`Psr\EventDispatcher\ListenerProviderInterface`

Esta interface é um mapeador de um evento para os `Listeners` que são aplicáveis a esse evento.

```
<?php
namespace Psr\EventDispatcher;

interface ListenerProviderInterface
{
    public function getListenersForEvent(object $event) : iterabl
;
}
```

`Psr\EventDispatcher\StoppableEventInterface`

Esta interface verifica um evento cujo processamento pode ser interrompido quando o evento foi tratado.

```
<?php
namespace Psr\EventDispatcher;

interface StoppableEventInterface
{
    public function isPropagationStopped() : bool;
}
```

Conclusão

No próximo capítulo vamos abordar sobre a PSR-15, que trata conceitos conhecidos como os `middlewares` / `handlers` . Então vamos nessa!

Capítulo 14

PSR-15 (HTTP SERVER REQUEST HANDLERS)

Com o principal objetivo de fornecer interfaces comuns para a manipulação das requisições HTTP, os `handlers` (manipuladores) são essenciais em qualquer aplicação Web. O código do lado do servidor recebe uma requisição que é processada e então é gerada uma resposta que pode ser repassada para outra aplicação ou não.

Os `middlewares` em APIs podem ser considerados como estruturas que trabalham sob o protocolo HTTP, por meio do qual recebem uma requisição de entrada e geram uma resposta como saída. Essa resposta pode ou não ser repassada para outros middlewares.

Por exemplo, para realizar a autenticação de um usuário, teríamos algo como um middleware de autenticação que recebe as credenciais do usuário. Se os dados estiverem corretos, ele poderá passar para o próximo middleware, indicando que a autenticação foi bem-sucedida e ele está pronto para receber autorização para acessar os recursos permitidos para aquele usuário. Para entendermos melhor o funcionamento dos middlewares, veja o exemplo a seguir, que demonstra o seu funcionamento dentro de

uma API:

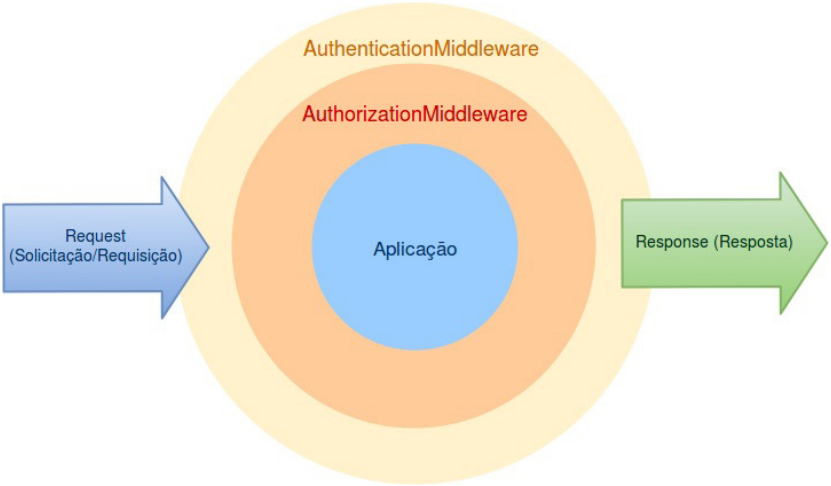


Figura 14.1: Funcionamento dos middlewares

Conforme mostrado na imagem anterior nós temos duas camadas de middlewares. O `AuthenticationMiddleware` recebe a requisição e realiza a verificação concedendo ou não o acesso ao usuário. Em caso afirmativo, os dados da requisição são passados para o próximo middleware, o `AuthorizationMiddlware` , que é responsável por verificar se o usuário possui autorização para acessar um determinado recurso do sistema. Se sim, então os dados são enviados para a camada da aplicação para continuar o processamento. Podemos ter quantos middlewares forem necessários, tudo vai depender da sua aplicação.

14.1 REGRAS

Vamos conhecer a seguir as regras que definem essa PSR para entendermos como aplicá-la de forma coesa.

Regra 1

Request Handlers (Manipuladores de Requisição/Solicitação) - É um componente individual que realiza o processamento de uma solicitação e consequentemente gera uma resposta, conforme definido pela PSR-7. Um `request handler` (manipulador de requisição) PODE lançar uma exceção caso as condições da requisição sejam impedidas de produzirem uma resposta, porém, o tipo de exceção não está definido.

Os `requests handlers` que utilizam esse padrão DEVEM implementar a interface `Psr\Http\Server\RequestHandlerInterface`.

Regra 2

Middleware - É um componente individual que trabalha junto com outros middlewares durante o processamento de uma requisição e gera uma resposta como saída, conforme definido pela PSR-7.

Um middleware PODE criar e retornar uma resposta sem que haja a delegação para um `request handler` (manipulador de requisição) desde que as condições necessárias sejam atendidas.

Para utilizar este padrão, os middlewares DEVEM implementar a interface `Psr\Http\Server\MiddlewareInterface`.

Regra 3

Generating Responses (Gerando Respostas) - É RECOMENDADO que qualquer manipulador de middleware ou

de requisição que gere uma resposta componha um modelo de resposta baseado na PSR-7, implementando a interface `Psr\Http\Message\ResponseInterface` ou até mesmo uma `factory` (fábrica) capaz de gerar uma instância de `Psr\Http\Message\ResponseInterface`, evitando a dependência de uma implementação específica de mensagem HTTP.

Regra 4

Handling Exceptions (Tratamento de Exceções) - É RECOMENDADO que qualquer aplicação que utilizar o middleware inclua um componente responsável por realizar a captura das exceções e as converta em respostas. Este middleware DEVE ser o primeiro componente a ser executado e envolver todo o processamento adicional para garantir que uma resposta seja sempre gerada.

Essas são as regras que a PSR-15 define. Não é nada complicado e é tecnicamente muito simples de implementar em sua aplicação. A maioria dos frameworks e microframeworks já possui suporte a essa PSR, basta você desenvolver a lógica da sua aplicação e seguir as regras conforme estipuladas, ok?

14.2 INTERFACES

A seguir vamos conhecer as interfaces que fazem parte desse pacote.

`Psr\Http\Server\RequestHandlerInterface`

```
<?php
```

```
namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): Resp
onseInterface;
}
```

Psr\Http\Server\MiddlewareInterface

```
<?php

namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    public function process(ServerRequestInterface $request, Requ
estHandlerInterface $handler): ResponseInterface;
}
```

Conclusão

Conforme você pôde ver, são duas interfaces bem simples que não possuem muitos métodos e com isso chegamos ao final da PSR-15 que definem os conceitos de middlewares e manipuladores de requisição para que sua aplicação possa ou não delegar os dados para um outro middleware.

Em nosso próximo capítulo vamos falar da PSR-16, que define interfaces comuns para aplicações de cache, vamos nessa!

Capítulo 15

PSR-16 (COMMON INTERFACE FOR CACHING LIBRARIES)

Essa PSR possui o objetivo de descrever uma interface simples, porém extensível para um item de cache e um driver de cache, sendo que as implementações finais PODEM decorar os objetos com mais funcionalidades do que o que foi proposto, mas estes DEVEM realizar a implementação das funcionalidades da interface primeiro.

O cache é uma maneira de melhorar o desempenho de qualquer aplicação, tornando aplicações/bibliotecas de cache uma das características mais comuns de muitos frameworks. Obter interoperabilidade neste nível significa que podemos realizar a implementação de cache específico para a aplicação e confiar no que é enviado ao cache pelo framework ou qualquer outra biblioteca de cache.

É importante mencionar que a PSR-6 já atende perfeitamente esse problema, porém de maneira bem detalhada. O que é abordado aqui na PSR-16 é uma maneira mais simples que visa construir uma interface simplificada padronizada para os casos

comuns, além disso, é independente da PSR-6 e foi projetada para ter compatibilidade com ela de maneira simples.

15.1 REGRAS

Vamos conhecer a seguir a regras que a PSR-16 define para entendermos melhor.

Regra 1

Calling Library (Chamando na Aplicação) - A aplicação ou o código que realmente precisar utilizar os serviços de cache deverá implementar a interface padrão `Psr\SimpleCache\CacheInterface` , caso contrário, não saberá da existência do serviço de cache.

Regra 2

Implementing Library (Implementando na Aplicação) - A aplicação é responsável por implementar o padrão estabelecido pela PSR com o objetivo de fornecer os serviços de cache para qualquer aplicação ou código de forma simples. A implementação DEVE prover a classe que implementa a interface `Psr\SimpleCache\CacheInterface` . A implementação de bibliotecas DEVE suportar ao menos a funcionalidade `TTL (Time To Live)` .

Regra 3

TTL (Time To Live) - Cada item de cache possui o que chamamos de tempo de vida, esse tempo é o que define se

determinado item está obsoleto ou não. O `TTL` é definido por um número inteiro que representa o tempo em segundos ou por um objeto `DateInterval`.

Regra 4

Expiration (Expiração) - É o tempo definido para que o cache se torne obsoleto, geralmente é calculado adicionando o `TTL (Time To Live)` ao tempo em que um determinado objeto é armazenado. O tempo também pode ser adicionado através de um objeto `DateTime`, por exemplo, um item de cache com um `TTL (Time To Live)` de 300 segundos que foi armazenado às 09:00:00 terá um tempo de expiração em 09:05:00, ou seja, terá um tempo de 5 minutos antes de tornar-se obsoleto.

A aplicação PODE expirar um item antes de seu tempo de expiração definido, porém DEVE tratar um determinado item como expirado quando o seu tempo de expiração for atingido. Se a aplicação passar um item para ser armazenado e não especificar um tempo de expiração ou `TTL (Time To Live)`, ela PODE utilizar um tempo de expiração padrão.

Caso nenhum tempo de duração for definido, então a aplicação DEVE interpretar como uma solicitação para armazenar o item em cache para sempre ou pelo tempo em que a implementação suportar.

Regra 5

Key (Chave) - É a chave que define um item de cache, e esse nome deve ser exclusivo para cada item de cache. As bibliotecas DEVEM suportar as chaves que consistam nos caracteres de A-Z,

a-z, 0-9, undescore (_) e ponto (.), e em qualquer ordem na codificação UTF-8, sendo que o nome da chave deve ter um comprimento de no máximo 64 caracteres.

A aplicação que implementar PODE suportar outros caracteres, outras codificações e/ou um nome de chave com um comprimento maior. A aplicação também será responsável por escapar a sequência de caracteres conforme a necessidade, mas DEVE retornar a sequência original da chave sem nenhum tipo de modificação.

Os caracteres `{` , `}` , `(` , `)` , `/` , `\` , `@` , `:` NÃO DEVEM ser suportados pela aplicação, isso porque são reservados para extensões futuras.

Regra 6

Cache - Um objeto que implementa a interface `Psr\SimpleCache\CacheInterface` .

Regra 7

Cache Misses - Um erro de cache retornará `null` (nulo), portanto, detectar se um valor nulo está armazenado é impossível.

Regra 8

A aplicação que realizar a implementação PODE fornecer um mecanismo para que o desenvolvedor possa especificar um padrão `TTL` - Caso nenhum valor seja especificado para um item de cache, a aplicação DEVE utilizar como valor padrão o valor máximo permitido pela implementação subjacente. Caso a

implementação subjacente não possua suporte ao `TTL`, o valor especificado pelo desenvolvedor para o `TTL` DEVE ser ignorado.

Regra 9

A aplicação DEVE suportar todos os tipos de dados PHP que possam ser serializados - Isso inclui tipos como:

- **`Strings`** - Cadeia de caracteres de qualquer tamanho e em qualquer codificação compatível com o PHP.
- **`Integers`** - Todos os inteiros de qualquer tamanho suportados pelo PHP, até 64 bits.
- **`Floats`** - Todos os pontos flutuantes.
- **`Boolean`** - `true` (verdadeiro) e `false` (falso).
- **`Null`** - O valor nulo, embora não seja possível distinguir de um erro de cache ao realizar a leitura.
- **`Arrays`** - Arrays indexados, associativos e multidimensionais de qualquer profundidade.
- **`Object`** - Qualquer objeto que tenha suporte a serialização e desserialização sem perdas, tal que `$object == unserialize(serialize($object))`. Objetos PODEM aproveitar os métodos mágicos `sleep()` ou `wakeup()` do PHP.

Regra 10

Todos os dados passados para a aplicação DEVEM ser retornados da mesma maneira como foram passados. Isso significa que, se a aplicação passar uma variável com o valor (string) "10", ela DEVERÁ retornar o valor (string) "10", e não um valor (int) 10.

A aplicação PODE utilizar as funções `serialize()` e `unserialize()` do PHP, mas não é obrigatório utilizá-las. A compatibilidade é simplesmente usada como uma base para objetos que são aceitos. Se por algum motivo o valor exato não puder ser retornado, a aplicação DEVE retornar um erro de cache e não um erro de dados corrompidos/inválidos.

Conhecemos as regras que a PSR-16 define e como você pôde perceber são bem semelhantes com as regras da PSR-6. A diferença é que a PSR-16 é um modelo mais simples e enxuto da PSR-6, isso não impede que você utilize as duas PSRs em conjunto.

15.2 INTERFACES

A seguir vamos conhecer as interfaces que fazem parte dessa PSR.

`Psr\SimpleCache\CacheInterface`

Esta interface define as operações básicas para uma coleção de cache, que requer leitura, gravação e exclusão de itens de cache individuais, além disso, possui métodos que lidam com várias entradas de cache, como escrever, ler ou excluir várias entradas de cache por vez. Isso é muito útil quando temos muitas leituras/gravações em cache para executar e permite executar essas operações através de uma única chamada para o servidor de cache, reduzindo drasticamente a latência.

Uma única instância de `Psr\SimpleCache\CacheInterface` é correspondente a uma única coleção de itens de cache com um único namespace de chave e é equivalente a um `pool` da PSR-6. É permitido diferentes instâncias de

`Psr\SimpleCache\CacheInterface` e essas PODEM ser apoiadas pelo mesmo armazenamento de dados, porém, DEVEM ser logicamente independentes. A definição dessa interface é mostrada a seguir:

```
<?php

namespace Psr\SimpleCache;

interface CacheInterface
{
    public function get($key, $default = null);
    public function set($key, $value, $ttl = null);
    public function delete($key);
    public function clear();
    public function getMultiple($keys, $default = null);
    public function setMultiple($values, $ttl = null);
    public function deleteMultiple($keys);
    public function has($key);
}
```

`Psr\SimpleCache\CacheException`

Esta interface define as exceções de cache que podem ocorrer.

```
<?php

namespace Psr\SimpleCache;

interface CacheException
{
}
```

`Psr\SimpleCache\InvalidArgumentException`

Esta interface define as exceções de argumentos inválidos que podem ocorrer.

```
<?php

namespace Psr\SimpleCache;
```

```
interface InvalidArgumentException extends CacheException
{
}
```

Conclusão

A PSR-16 define uma interface comum para trabalhar com cache de maneira simples e compatível também com a PSR-6, que também trabalha com cache.

No próximo capítulo vamos falar sobre a PSR-17, que trabalha `HTTP Factory` para a criação de objetos definidos pela PSR-7.

Capítulo 16

PSR-18 (HTTP CLIENT)

Esta PSR descreve uma interface comum para o envio e recebimento de requisições `HTTP` compatíveis com a PSR-7. Um dos principais objetivos é permitir que os desenvolvedores criem bibliotecas totalmente desacopladas das implementações do cliente `HTTP`, tornando-as ainda mais reutilizáveis. Além disso, o intuito desta PSR é diminuir a quantidade de dependências e ainda diminuir a probabilidade de conflitos de versões.

Um outro ponto importante é que os clientes `HTTP` possam ser substituídos de acordo com o **princípio de substituição de Liskov**. Todos os clientes, ao enviarem uma requisição, DEVEM se comportar exatamente da mesma maneira.

O QUE É O PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV (LSP)?

Barbara Liskov criou uma definição para o conceito de subtipo. Segundo Liskov: Se q(x) é uma propriedade demonstrável dos objetos x de tipo T. Então q(y) deve ser verdadeiro para objetos y de tipo S onde S é um subtipo de T.

De maneira bem resumida, o princípio de Liskov sugere que uma classe base deve poder ser substituída pela sua classe derivada.

16.1 REGRAS

Vamos conhecer a seguir as regras que fazem parte dessa PSR.

Regra 1

Um cliente PODE:

- Optar por enviar uma requisição `HTTP` diferente da requisição que foi fornecida. Um exemplo seria a compactação do corpo de uma mensagem de resposta.
- Optar por alterar uma resposta `HTTP` recebida antes de retorná-la para quem a solicitou. Um exemplo seria a descompactação de um corpo de mensagem de entrada.

Regra 2

Se um cliente optar por alterar a requisição ou a resposta `HTTP` , o cliente DEVE garantir que o objeto permaneça

internamente consistente. Por exemplo, ao descompactar o corpo da mensagem o cliente também DEVERÁ remover o cabeçalho `Content-Encoding` e ajustar o cabeçalho `Content-Length`.

Regra 3

Como os objetos definidos pela PSR-7 são imutáveis, a aplicação NÃO DEVE assumir que o objeto enviado para `ClientInterface::sendRequest()` será realmente o mesmo objeto enviado. Por exemplo, o objeto `Request` que é retornado por uma exceção PODE ser diferente daquele que foi passado para `sendRequest()`, portanto, a comparação por referência (`===`) não será possível.

Regra 4

Um cliente DEVE remontar uma resposta `HTTP` de status 1xx (informativa), ou seja, com os status (100 - Continuar, 101 - Mudando Protocolos, 102 - Processamento e 122 - Pedido-URI Muito Longo) de várias etapas para que o que é retornado para a aplicação seja uma resposta `HTTP` válida do código de status 200 ou superior.

Regra 5

Um cliente NÃO DEVE tratar uma requisição `HTTP` bem formada ou uma resposta `HTTP` como uma condição de erro. Os códigos de status de resposta na faixa 400 e 500 NÃO PODEM causar uma exceção e DEVEM ser devolvidos para a aplicação normalmente.

Regra 6

Um cliente deve lançar uma exceção de `Psr\Http\Client\ClientExceptionInterface` se, e somente se, o cliente não puder enviar uma requisição `HTTP` ou se a resposta `HTTP` não puder ser analisada em um objeto de resposta da PSR-7.

Regra 7

Caso uma requisição não possa ser enviada porque a mensagem de requisição não é uma requisição `HTTP` bem formada ou está faltando alguma informação crítica (como um host ou método), o cliente DEVE lançar uma exceção de `Psr\Http\Client\RequestExceptionInterface`.

Regra 8

Se a requisição não puder ser enviada devido a uma falha de rede de qualquer tipo, incluindo um tempo limite, o cliente DEVE lançar uma exceção de `Psr\Http\Client\NetworkExceptionInterface`.

Regra 9

Os clientes PODEM lançar exceções mais específicas do que as que foram definidas por esta PSR (um `TimeOutException` ou `HostNotFoundException`, por exemplo), desde que implementem a interface apropriada definida anteriormente.

Essas foram as regras definidas pela PSR-18 e, como pudemos

ver, não são muitas e também não são tão complexas como outras regras que vimos nas PSRs anteriormente.

16.2 INTERFACES

A seguir vamos conhecer as interfaces que fazem parte dessa PSR.

`Psr\Http\Client\ClientInterface`

Esta interface implementa apenas um método e possui a responsabilidade de enviar e receber uma requisição compatível com a PSR-7.

```
<?php
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

interface ClientInterface
{
    public function sendRequest(RequestInterface $request): Respo
nseInterface;
}
```

`Psr\Http\Client\ClientExceptionInterface`

Toda exceção relacionada ao cliente `HTTP` DEVE implementar esta interface.

```
<?php
namespace Psr\Http\Client;

interface ClientExceptionInterface extends \Throwable
{
}
```

`Psr\Http\Client\RequestExceptionInterface`

Esta interface possui a responsabilidade de lançar uma exceção para o caso de uma requisição falhar.

```
<?php
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

interface RequestExceptionInterface extends ClientExceptionInterf
ace
{
    public function getRequest(): RequestInterface;
}
```

`Psr\Http\Client\NetworkExceptionInterface`

Esta interface possui a responsabilidade de lançar uma exceção quando a requisição não puder ser concluída devido a problemas de rede.

```
<?php
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

interface NetworkExceptionInterface extends ClientExceptionInterf
ace
{
    public function getRequest(): RequestInterface;
}
```

Conclusão

Como pudemos ver, trata-se de uma PSR bem intuitiva e que demonstra como podemos ter uma interface comum para o envio e recebimento de requisições `HTTP` compatíveis com a PSR-7, tornando as bibliotecas ainda mais reutilizáveis e reduzindo

drasticamente a quantidade de dependências.

CAPÍTULO 17

PSR-17 (HTTP FACTORIES)

A PSR-17 define interfaces que são utilizadas para fabricar objetos `HTTP` definidos pela PSR-7, ou seja, a PSR-17 é uma `HTTP Factory` (fábrica de objetos `HTTP`) provendo interfaces para essas fábricas. Sendo assim, as fábricas `HTTP` DEVEM implementar as interfaces providas pela PSR-17 para cada tipo de objeto que é fornecido pela PSR-7.

Diferentemente das PSRs anteriores, esta não possui um conjunto de regras a serem seguidas, mas sim um conjunto de interfaces que devem ser implementadas. E são elas que vamos conhecer na próxima seção.

17.1 INTERFACES

As interfaces a seguir PODEM ser implementadas de duas maneiras: juntas em uma única classe ou em classes separadas.

`Psr\Http\Message\RequestFactoryInterface`

Esta interface possui a responsabilidade de criar requisições `HTTP`.

```
<?php
namespace Psr\Http\Message;
```

```
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\UriInterface;

interface RequestFactoryInterface
{
    public function createRequest(string $method, $uri): RequestI
nterface;
}
```

`Psr\Http\Message\ResponseFactoryInterface`

Esta interface possui a responsabilidade de criar respostas `HTTP`.

```
<?php
namespace Psr\Http\Message;

use Psr\Http\Message\ResponseInterface;

interface ResponseFactoryInterface
{
    public function createResponse(int $code = 200, string $reaso
nPhrase = ''): ResponseInterface;
}
```

`Psr\Http\Message\ServerRequestFactoryInterface`

Esta interface possui a responsabilidade de criar requisições do servidor.

```
<?php
namespace Psr\Http\Message;

use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\UriInterface;

interface ServerRequestFactoryInterface
{
    public function createServerRequest(string $method, $uri, arr
ay $serverParams = []): ServerRequestInterface;
}
```

Psr\Http\Message\StreamFactoryInterface

Esta interface possui a responsabilidade de criar fluxos para requisições e respostas.

```
<?php
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;

interface StreamFactoryInterface
{
    public function createStream(string $content = ''): StreamInt
erface;

    public function createStreamFromFile(string $filename, string
$mode = 'r'): StreamInterface;

    public function createStreamFromResource($resource): StreamIn
terface;
}
```

As implementações desta interface DEVEM utilizar um fluxo temporário ao criar recursos de cadeia de caracteres. A maneira recomendada para fazer isso é a demonstrada no código a seguir:

```
<?php
$resource = fopen('php://temp', 'r+');
```

Psr\Http\Message\UploadedFileFactoryInterface

Esta interface possui a responsabilidade de criar fluxos para arquivos enviados, ou seja, para uploads de arquivos.

```
<?php
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;
use Psr\Http\Message\UploadedFileInterface;

interface UploadedFileFactoryInterface
```

```
{
    public function createUploadedFile(
        StreamInterface $stream,
        int $size = null,
        int $error = \UPLOAD_ERR_OK,
        string $clientFilename = null,
        string $clientMediaType = null
    ): UploadedFileInterface;
}
```

`Psr\Http\Message\UriFactoryInterface`

Esta interface possui a responsabilidade de criar URIs para requisições de clientes e servidores.

```
<?php
namespace Psr\Http\Message;

use Psr\Http\Message\UriInterface;

interface UriFactoryInterface
{
    public function createUri(string $uri = '') : UriInterface;
}
```

Conclusão

Apesar de esta PSR não ser grande e não possuir definições de regras, as interfaces descritas anteriormente possuem a capacidade de criar objetos `HTTP` definidos pela PSR-7.

No próximo capítulo veremos a PSR-18, que descreve interface para enviar e receber requisições `HTTP` .

Capítulo 18

CONCLUSÃO

Como pudemos ver durante este livro, cada PSR define um conjunto de definições e possui as suas características. Vimos como deixar o nosso código mais legível com a PSR-2, como podemos implementar um padrão para a escrita de logs com a PSR-3, vimos ainda como padronizar nossas requisições com a PSR-7 e muitas outras boas práticas que poderão tornar a vida dos desenvolvedores mais fácil.

Por fim, espero que tenha gostado deste livro e que você tenha obtido um excelente conhecimento de boas práticas de programação com o PHP através da PHP-FIG e as PSRs.

Boa sorte no mundo da programação em PHP, um grande abraço e até a próxima!

Capítulo 19

REFERÊNCIAS BIBLIOGRÁFICAS

BOGGIANO, Jordi. PSR-3 (Logger Interface). Disponível em: https://www.php-fig.org/psr/psr-3 /. Acesso em: 01 de outubro de 2018.

DRAGOONIS, Paul. PSR-16 (Simple Cache). Disponível em: https://www.php-fig.org/psr/psr-16/. Acesso em: 17 de fevereiro de 2019.

GARFIELD, Larry. PSR-6 (Caching Interface). Disponível em: https://www.php-fig.org/psr/psr-6/. Acesso em: 15 de outubro de 2018.

GARFIELD, Larry. PSR-13 (Hypermedia Links). Disponível em: https://www.php-fig.org/psr/psr-13/. Acesso em: 03 de dezembro de 2018.

GARFIELD, Larry. PSR-14 (Event Dispatcher). Disponível em: https://www.php-fig.org/psr/psr-14/. Acesso em: 17 de abril de 2019.

GILK, Woody. PSR-15 (HTTP Handlers). Disponível em: https://www.php-fig.org/psr/psr-15/. Acesso em: 05 de janeiro de

2019.

GILK, Woody. PSR-17 (HTTP Factories). Disponível em: https://www.php-fig.org/psr/psr-17/. Acesso em: 09 de março de 2019.

JONES, Paul M. PSR-1 (Basic Coding Standard). Disponível em: https://www.php-fig.org/psr/psr-1/. Acesso em: 20 de setembro de 2018.

JONES, Paul M. PSR-2 (Coding Style Guide). Disponível em: https://www.php-fig.org/psr/psr-2/. Acesso em: 21 de setembro de 2018.

JONES, Paul M. PSR-4 (Autoloading Standard). Disponível em: https://www.php-fig.org/psr/psr-4/. Acesso em: 01 de outubro de 2018.

NAPOLI, Matthieu e NÉGRIER, David. PSR-11 (Container Interface). Disponível em: https://www.php-fig.org/psr/psr-11/. Acesso em: 15 de novembro de 2018.

NYHOLM, Tobias. PSR-18 (HTTP Client). Disponível em: https://www.php-fig.org/psr/psr-18/. Acesso em: 17 de abril de 2019.

O'PHINNEY, Matthew Weier. PSR-7 (HTTP Message Interface). Disponível em: https://www.php-fig.org/psr/psr-7/. Acesso em: 12 de novembro de 2018.