

SUMÁRIO

1. GERENCIANDO DEPENDÊNCIAS COM COMPOSER.....	1
2. INSTALAÇÃO DO COMPOSER.....	1
2.1. BAIXANDO O COMPOSER.....	1
2.1.1. VERIFICANDO A INSTALAÇÃO.....	1
3. UTILIZANDO O COMPOSER.....	1
3.1. ADICIONANDO DEPENDÊNCIAS.....	3
3.2. UTILIZANDO AS DEPENDÊNCIAS EXTERNAS NO PROJETO	4
3.3. UTILIZANDO DEPENDÊNCIAS CRIADAS.....	5
3.3.1. CRIANDO A DEPENDÊNCIA INTERNA.....	5
3.3.2. UTILIZANDO A DEPENDÊNCIA CRIADA.....	6
4. ESCLARENDO DÚVIDAS.....	7
4.1. FUNCIONAMENTO DO USE E DO NAMESPACE.....	7
4.2. SÍNTEXE DO USE	7
5. FONTES.....	8

1. GERENCIANDO DEPENDÊNCIAS COM COMPOSER

O Composer é um gerenciador de pacotes no nível do aplicativo para a linguagem de programação PHP que fornece um formato padrão para gerenciar dependências do software PHP e bibliotecas necessárias.

Ou seja, o Composer serve para gerenciar os dados que, em alguma instância, na aplicação, são dependências, isto é, necessários para funcionalidades e mecanismos implementados. A exemplo, uma dependência poderia ser uma série de métodos de uma Classe X, desenvolvida por um autor Y que a publicou no [Packagist](#), e assim, utilizando o Composer, podemos baixa-la, utiliza-la e gerencia-la, dentro do projeto, com o Composer.

2. INSTALAÇÃO DO COMPOSER

2.1. BAIXANDO O COMPOSER

[Como diz no site para download do Composer](#), o instalador - que requer que você já tenha o PHP instalado - fará o download do Composer para você e configurará sua variável de ambiente PATH para que você possa simplesmente chamar composer de qualquer diretório.

2.1.1. VERIFICANDO A INSTALAÇÃO

Como diz no site, ao instalar o Composer é criada uma variável de ambiente para que as funcionalidades possam ser acessadas a partir de qualquer diretório.

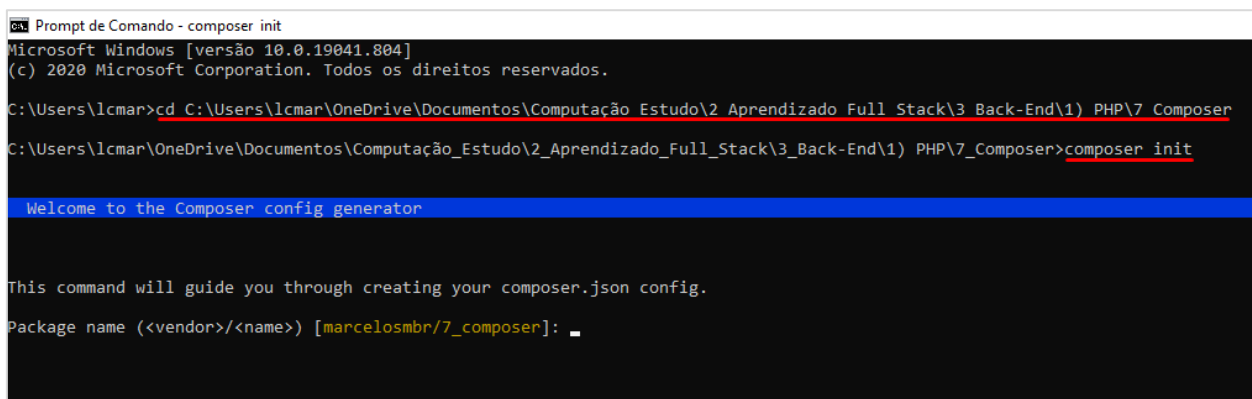
Assim, para verificar se está instalado, devemos acessar o CMD e digitar “composer -V”. Se de fato a instalação tiver sido realizada, será impresso na tela a versão atual do programa.

3. UTILIZANDO O COMPOSER

Primeiramente, o arquivo base do Composer é o que alguns denominam como “manifesto”, pois é um arquivo de configuração que serve para definir um conjunto de informações e dados para o funcionamento do programa em si e das circunstâncias presentes do projeto.

Este arquivo, cuja extensão é .json, pode ser criado manualmente, ou gerado automaticamente a partir da linha de comando e em um diretório especificado – que é um método preferível. Portanto, para isso, na linha de comando, acessaremos o diretório raiz do projeto, com “cd caminho”, e digitaremos ‘composer init’.

Figura 1 Composer init



```
CA Prompt de Comando - composer init
Microsoft Windows [versão 10.0.19041.804]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.

C:\Users\lcmar>cd C:\Users\lcmar\OneDrive\Documentos\Computação Estudo\2_Aprendizado_Full_Stack\3_Back-End\1) PHP\7 Composer
C:\Users\lcmar\OneDrive\Documentos\Computação_Estudo\2_Aprendizado_Full_Stack\3_Back-End\1) PHP\7_Composer>composer init

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.
Package name (<vendor>/<name>) [marcelosmbr/7_composer]: _
```

Fonte: Autor

Com isso, como podemos ver acima, se dá início ao processo para gerar um manifesto do Composer. A configuração do composer.json se dá em uma série de questões que devem ser respondidas de acordo com as circunstâncias do projeto.

Por exemplo, a primeira questão é sobre o nome do pacote que, por via de regra, deve ser escrito no formato `vendor/nome_projeto`, onde `vendor` é quem está distribuindo esse pacote (como um nickname, um nome de usuário do github ou o nome da empresa).

Em seguida, na segunda questão teremos que passar uma descrição do pacote; na terceira questão teremos que informar o autor, que, na maioria das vezes, por default é considerado um email; na quarta questão “minimum-stability” deve ser informada a versão da aplicação, podendo ser “dev”, isto é, versão de desenvolvimento, “alpha”, “beta”, “stable” e outros; a quinta questão pede para informar o tipo de pacote, ou seja, se é um projeto, uma biblioteca, um plugin, etc; a sexta questão pede para informar a licença do pacote, se existir, e as posteriores irão variar dependendo das respostas.

Ou seja, estas questões estão definindo aspectos sobre a aplicação para o seu próprio funcionamento e para o caso em que seja publicada.

Abaixo vemos um arquivo `composer.json` criado sem dependências externas especificadas, mas que poderiam ter sido descritas no processo de geração do `composer.json`, e neste caso apareciam no atributo “require”.

Figura 2 Composer.json



```

1  {
2      "name": "marcelosmbr/composer_teste",
3      "description": "Primeiro Composer",
4      "type": "project",
5      "minimum-stability": "dev",
6      "require": []
7      //DEPENDÊNCIAS
8  }
9
10

```

Fonte: Autor

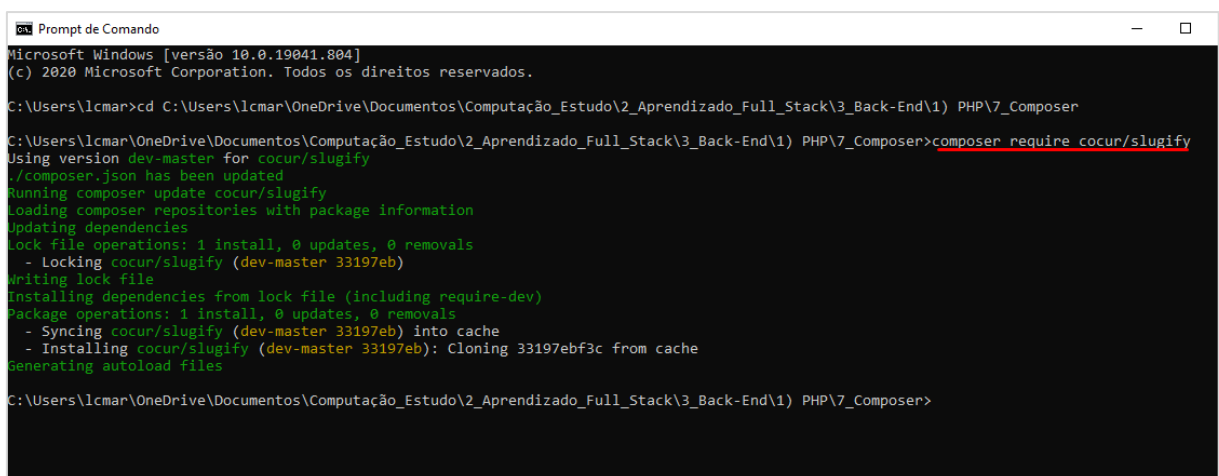
3.1. ADICIONANDO DEPENDÊNCIAS

Vamos adicionar uma dependência externa que será definida no atributo “require”, e esta será: <https://packagist.org/packages/cocur/sluggify>

Neste link, do site Packagist, há um pacote que traz consigo funcionalidades para transformar strings em um slug, que é o texto que descreve o caminho até o conteúdo, e que aparece logo após o domínio de uma URL.

Como diz na página de download do pacote, devemos escrever no CMD “composer require cocur/sluggify”, após, é claro, navegarmos até o diretório raiz do projeto.

Figura 3 Composer require



```

Microsoft Windows [versão 10.0.19041.804]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.

C:\Users\lcmar>cd C:\Users\lcmar\OneDrive\Documentos\Computação_Estudo\2_Aprendizado_Full_Stack\3_Back-End\1) PHP\7_Composer

C:\Users\lcmar\OneDrive\Documentos\Computação_Estudo\2_Aprendizado_Full_Stack\3_Back-End\1) PHP\7_Composer>composer require cocur/sluggify
Using version dev-master for cocur/sluggify
./composer.json has been updated
Running composer update cocur/sluggify
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking cocur/sluggify (dev-master 33197eb)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Syncing cocur/sluggify (dev-master 33197eb) into cache
  - Installing cocur/sluggify (dev-master 33197eb): Cloning 33197ebf3c from cache
Generating autoload files
C:\Users\lcmar\OneDrive\Documentos\Computação_Estudo\2_Aprendizado_Full_Stack\3_Back-End\1) PHP\7_Composer>

```

Fonte: Autor

Após este passo, automaticamente o campo “require”, do composer.json, será preenchido, e, além disto, será criado outro arquivo, no diretório raiz do projeto, chamado

“**composer.lock**”, que registra as versões exatas das dependências que foram instaladas, e também um diretório chamado “**vendor**”, que armazena as dependências do projeto, bem como o imprescindível “**autoload**” que mapeia a aplicação, suas dependências e permite que sejam utilizadas.

Figura 4 Atributo "require"



```

{} composer.json X
7_Composer > {} composer.json > {} require
1  {
2      "name": "marcelosmbr/composer_teste",
3      "description": "Primeiro Composer",
4      "type": "project",
5      "minimum-stability": "dev",
6      "require": {
7          "cocur/slugify": "dev-master"
8      }
9  }
10

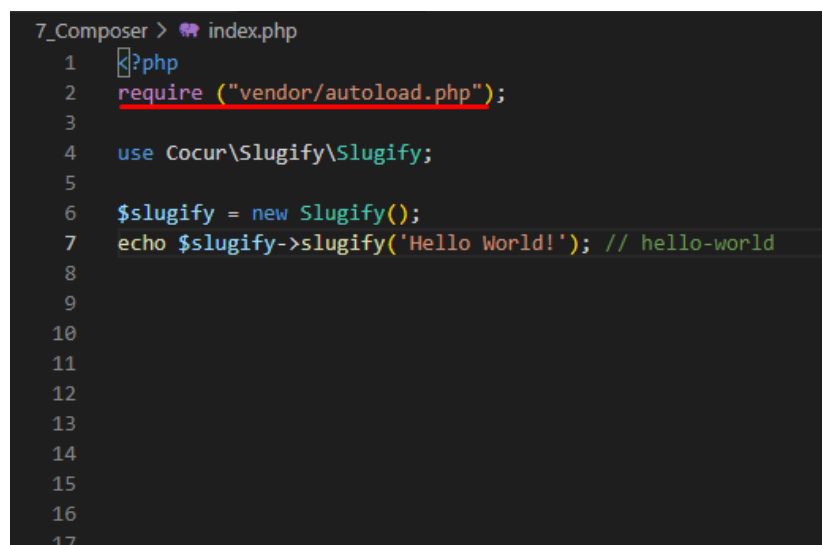
```

Fonte: Autor

3.2. UTILIZANDO AS DEPENDÊNCIAS EXTERNAS NO PROJETO

Devemos escrever, no topo do código fonte PHP, um comando require para requirir o **autoload**, para assim podermos utilizar as funcionalidades do Composer e da dependência externa instalada: `require("vendor/autoload.php")`. Em seguida, tendo como base o guia de uso da [dependência instalada](#), que transforma strings em slugs, para testar escrevemos:

Figura 5 Utilizando a dependência



```

7_Composer > index.php
1  <?php
2  require ("vendor/autoload.php");
3
4  use Cocur\Slugify\Slugify;
5
6  $slugify = new Slugify();
7  echo $slugify->slugify('Hello World!'); // hello-world
8
9
10
11
12
13
14
15
16
17

```

Fonte: Autor

3.3. UTILIZANDO DEPENDÊNCIAS CRIADAS

Para utilizar dependências ou classes criadas localmente, isto é, por nós mesmos, e não por terceiros, devemos adicionar no manifesto mais algumas especificações, e em seguida, no CMD digitar e executar o comando “**composer update**”, pois toda alteração realizada no manifesto deve ser seguida de um update do Composer.

Figura 6 Dependências internas

```

1  {
2      "name": "marcelosmbr/composer_teste",
3      "description": "Primeiro Composer",
4      "type": "project",
5      "minimum-stability": "dev",
6      "require": {
7          "cocur/slugify": "dev-master"
8      },
9      "autoload": {
10         "psr-4":{
11             "Classes\\": "class/"
12         }
13     }
14 }
15

```

Fonte: Autor

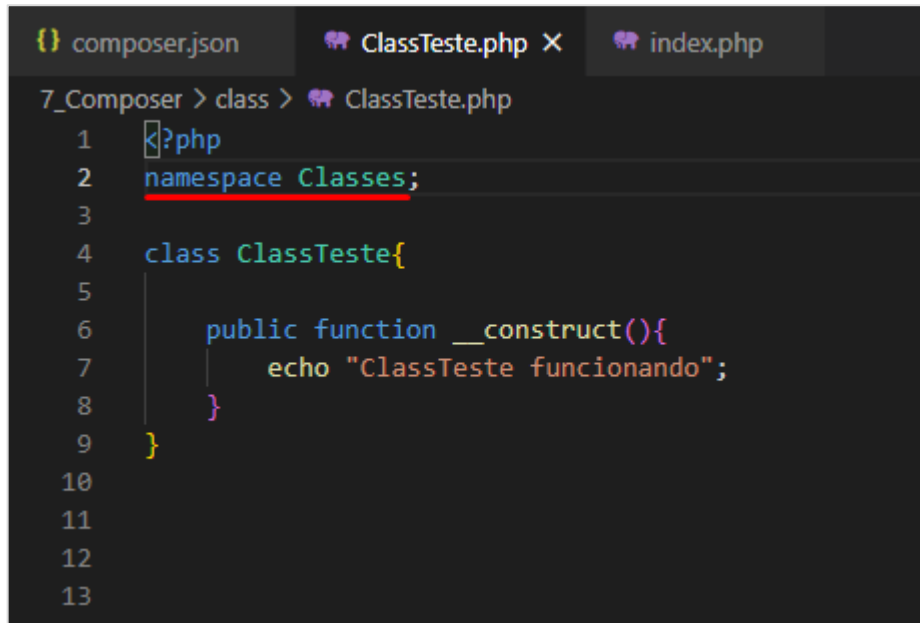
Portanto, dentro do atributo “**autoload**” são definidas as classes locais que devem ser gerenciadas para a utilização no projeto. Vale citar que o atributo **PSR-4** se refere ao padrão de desenvolvimento utilizado, e por via de regra, até agora, não deve mudar.

Em seguida, após a definição do atributo PSR-4, descrevemos as classes locais e informamos para o autoload onde e como acessa-las e mapeá-las. **O formato da descrição das classes é:** “Namespace”:”Diretório”.

3.3.1. CRIANDO A DEPENDÊNCIA INTERNA

No diretório raiz, criaremos uma pasta chamada “class”, e dentro dela um arquivo PHP “ClasseTeste.php” cujo código é a criação de uma classe chamada “ClasseTeste”, e de um namespace “Classes”, como de fato descrevemos para o autoload no manifesto: um namespace “Classes\\” que se refere ao contexto existente no diretório “class/”.

Figura 7 Criação da dependência interna



```

7_Composer > class > ClassTeste.php
1  <?php
2  namespace Classes;
3
4  class ClassTeste{
5
6      public function __construct(){
7          echo "ClassTeste funcionando";
8      }
9  }
10
11
12
13

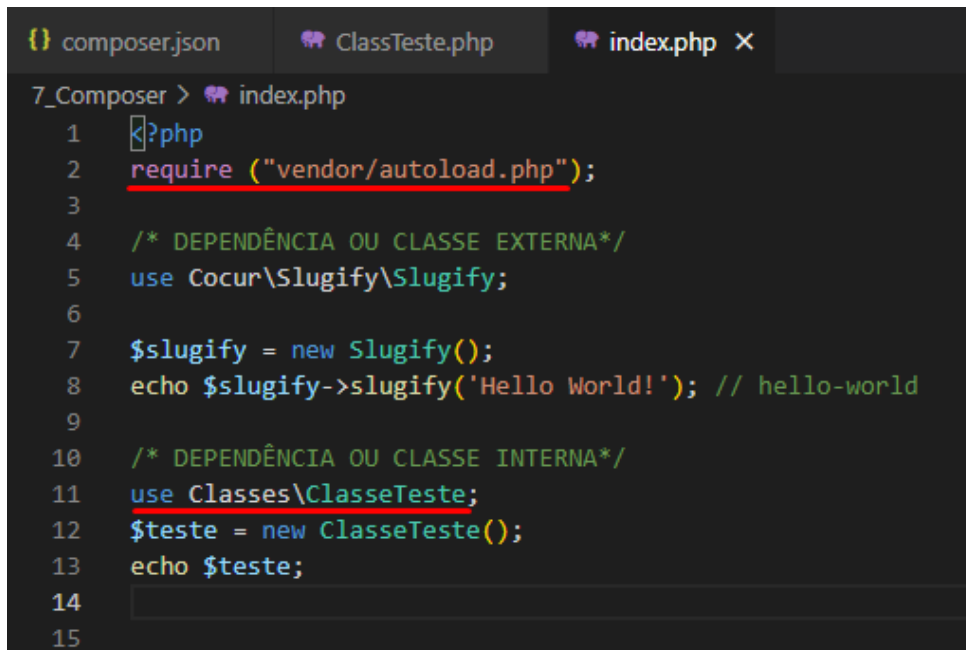
```

Fonte: Autor

3.3.2. UTILIZANDO A DEPENDÊNCIA CRIADA

Após criada a classe, a utilizamos no arquivo index como na imagem abaixo, e note que, obviamente, se entendido o funcionamento do Composer, com apenas uma requisição do “autoload”, todas as dependências, internas e externas, são mapeadas e ficam à disposição para uso na aplicação.

Figura 8 Utilizando a dependência interna



```

7_Composer > index.php
1  <?php
2  require ("vendor/autoload.php");
3
4  /* DEPENDÊNCIA OU CLASSE EXTERNA*/
5  use Cocur\Slugify\Slugify;
6
7  $slugify = new Slugify();
8  echo $slugify->slugify('Hello World!'); // hello-world
9
10 /* DEPENDÊNCIA OU CLASSE INTERNA*/
11 use Classes\ClasseTeste;
12 $teste = new ClasseTeste();
13 echo $teste;
14
15

```

Fonte: Autor

4. ESCLAREENDO DÚVIDAS

4.1. FUNCIONAMENTO DO USE E DO NAMESPACE

Imagine que existe uma classe “Teste”, existente em uma pasta “Class”, mapeada pelo autoload com o namespace “Classes”, e que aquela, quando instanciada, imprima uma mensagem pelo método construct.

Primeiro, não basta requisitar o autoload no arquivo index para ter acesso à classe “Teste”, deve-se utilizar, antes de instanciar a classe, o “use” do namespace definido no composer.json, seguido do nome factível da classe.

A requisição do autoload serve para importar suas configurações, que servem como ponte para os recursos, não sendo os próprios em si. **Já o “use” serve para** importar os próprios recursos a partir do namespace, ou “nome do caminho para os recursos”, definido nas configurações do autoload. Por isso a requisição deste último, do arquivo de autoload, antecede o “use”.

Segundo, se existente dentro de uma pasta mapeada uma classe X, isto é, em um caminho mapeado pelo autoload por meio de um namespace, a própria classe X deve ter o mesmo namespace. **O “namespace”** é exatamente como um nome reduzido e de acesso rápido. Quando definido um no autoload para uma pasta, por exemplo, estamos dizendo para “mapear este local, considerar seu nome de acesso como x, e garantir que este mesmo nome dê acesso também para os recursos que também o tenham”.

4.2. SÍNTEXE DO USE

Como bem sabemos, o “use”, no contexto de uso do composer, serve para importar os recursos mapeados por um namespace, e, agora adicionando a este entendimento, **é imprescindível que os nomes das classes sejam iguais aos nomes dos arquivos para a importação.**

Normalmente é comum que se crie, por exemplo, um arquivo de nome X, com extensão PHP, para o código de uma classe de nome semelhante, ou diferente, embora não seja recomendado por motivos de compreensibilidade da organização. Mas, no caso do uso do composer, [segundo a especificação PSR-4](#), o nome do arquivo de uma classe deve ser igual ao nome da própria.

Assim, a sintaxe da funcionalidade “use”, utilizada para importar os recursos mapeados por um namespace, é: **namespace\classname**, sendo “classname” igual ao nome da classe, e também do arquivo fonte.

5. FONTES

https://www.youtube.com/watch?v=hC6vVfb_C9U

<https://getcomposer.org/>

