

SUMÁRIO

1. O QUE É O COMPOSER	1
1.1. COMO INSTALAR (WINDOWS)	1
1.1.1. PACKAGIST.....	2
1.2. O MANIFESTO COMPOSER.JSON	2
1.2.1. CAMPO NAME	2
1.2.2. CAMPO DESCRIÇÃO	3
1.2.3. CAMPO TYPE	3
1.2.4. CAMPO MINIMUM-STABILITY	3
1.2.5. CAMPO REQUIRE.....	3
1.2.6. CAMPO AUTOLOAD	3
1.3. PSR: PHP STANDARDS RECOMMENDATIONS	3
2. COMO FUNCIONA O COMPOSER	4
2.1. INICIALIZANDO O COMPOSER NO PROJETO	4
2.2. INSTALANDO PACOTES EXTERNOS.....	5
2.3. MAPEANDO AS CLASSES INTERNAS: NAMESPACE E DUMP-AUTOLOAD	5
2.4. O AUTOLOAD E USO DAS CLASSES	6
2.4.1. UTILIZANDO AS CLASSES INTERNAS MAPEADAS	6
2.4.2. UTILIZANDO AS DEPENDÊNCIAS EXTERNAS INSTALADAS	6

1. O QUE É O COMPOSER

Composer é uma ferramenta para **gerenciamento de dependências em PHP**. Ele permite que você declare as bibliotecas das quais seu projeto depende e as gerenciara (instalará / atualizará) para você.

O Composer está para o PHP, assim como o NPM está para o Javascript. É uma ferramenta que funciona por projeto, e para gerenciar suas dependências. Assim, por padrão, e embora possa com comandos específicos, ele não instala nada globalmente, mas apenas localmente, na pasta de nome “vendor”, que é criada na sua inicialização.

Clique [aqui](#) para acessar o site oficial da tecnologia.

1.1. COMO INSTALAR (WINDOWS)

O primeiro passo é baixar o instalador executável do Composer para Windows no endereço <https://getcomposer.org/download/>. Depois de baixar o instalador, basta executá-lo e seguir as orientações.

A instalação cria uma variável de ambiente de nome “composer”. Para verificar se foi realizada corretamente basta digitar, sem aspas, “composer” no prompt de comando, ou no Power Shell.

```
C:\Users\lcmar>composer

Composer version 2.0.13 2021-04-27 13:11:08

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display this help message
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
      --ansi                Force ANSI output
      --no-ansi             Disable ANSI output
```

1.1.1. PACKAGIST

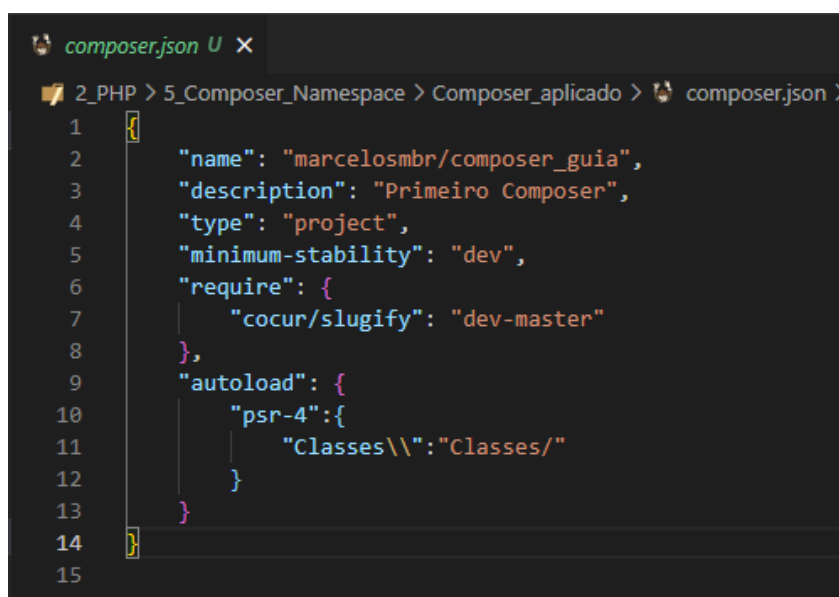
Packagist é o repositório de pacotes padrão do Composer. É de onde são obtidas as dependências externas, e onde ele mesmo procura pelos pacotes que requisitamos a instalação.

Também é possível, além de obter códigos desenvolvidos por terceiros, publicar pacotes para que outros utilizem. Para isso são necessários alguns passos, como configurações no arquivo manifesto, e que serão explicados no próximo tópico.

Clique [aqui](#) para acessar o site do Packagist.

1.2. O MANIFESTO COMPOSER.JSON

O pacote raiz, ou manifesto como foi chamado anteriormente, é o arquivo cujo nome é “composer.json”. Ele é gerado automaticamente quando o Composer é inicializado no projeto. É também chamado de “manifesto” porque expõe definições e requisitos do projeto.



```
1 {
2     "name": "marcelosmbr/composer_guia",
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\\": "Classes/"
12         }
13     }
14 }
```

Existem diversos outros campos possíveis além desses mostrados na imagem, mas a seguir serão descritos apenas esses, que estão dentre os frequentes. Os outros podem ser encontrados [aqui](#).

1.2.1. CAMPO NAME

É o nome do pacote. Consiste no nome do fornecedor e no nome do projeto, separados por /. Considerando o exemplo da imagem acima, “marcelosmbr” seria o nickname do autor, e “composer_guia” o nome do projeto.

O nome deve estar em letras minúsculas e consistir em palavras separadas por -, .ou _.

1.2.2. CAMPO DESCRIÇÃO

Uma breve descrição do pacote. Normalmente, esta é uma linha longa.

1.2.3. CAMPO TYPE

Os tipos de pacote são usados para lógica de instalação personalizada. Se você tem um pacote que precisa de alguma lógica especial, pode definir um tipo personalizado.

Por padrão o tipo é definido como “library”, mas existem outros 3, que são: project, metapackage e composer-plugin. Cada um possui uma especificação que pode ser conferida na [documentação](#).

1.2.4. CAMPO MINIMUM-STABILITY

Isso define o comportamento padrão para filtrar pacotes por estabilidade. O padrão é stable, portanto, se você depende de um pacote do ambiente de desenvolvimento (dev package), deve especificá-lo em seu arquivo.

Todas as versões de cada pacote são verificadas quanto à estabilidade e aquelas que são menos estáveis do que a “minimum-stability” serão ignoradas ao resolver as dependências do projeto.

As opções disponíveis (em ordem crescente de estabilidade) são: dev, alpha, beta, RC, e stable.

1.2.5. CAMPO REQUIRE

Esse campo é o mapeamento das dependências externas. Quando utilizado o comando para instalar pacotes de terceiros, elas serão definidas nesse campo – no formato nome e versão. Ou, também possível, elas podem ser escritas diretamente nesse campo, e após isso pode ser utilizado o comando para atualizar o manifesto.

De qualquer forma é para isso que esse campo serve – é o “mapa” dos pacotes externos.

1.2.6. CAMPO AUTOLOAD

Esse campo é o mapeamento dos recursos internos, como arquivos e classes. Como diz na documentação, é recomendando o uso das orientações da [PSR-4](#) para esse procedimento.

1.3. PSR: PHP STANDARDS RECOMMENDATIONS

A diferença no modo como os desenvolvedores escrevem código pode gerar uma verdadeira salada de frutas dependendo do tamanho da equipe e quantas pessoas já passaram no projeto ao longo do tempo.

Baseado nisso muitas linguagens e tecnologias definem recomendação de padrões. No PHP essas recomendações são criadas por um grupo chamado [PHP-FIG](#) e são chamadas de PSR.

As recomendações criadas pelo PHP-FIG são agrupadas em PHP Standard Recommendation (PSR). Uma PSR basicamente possui recomendações sobre um tema específico, como por exemplo, e que devem ser lidas, a PSR-1 que fala sobre padronização básica do código, e a PSR-12 que descreve o estilo como o código é escrito.

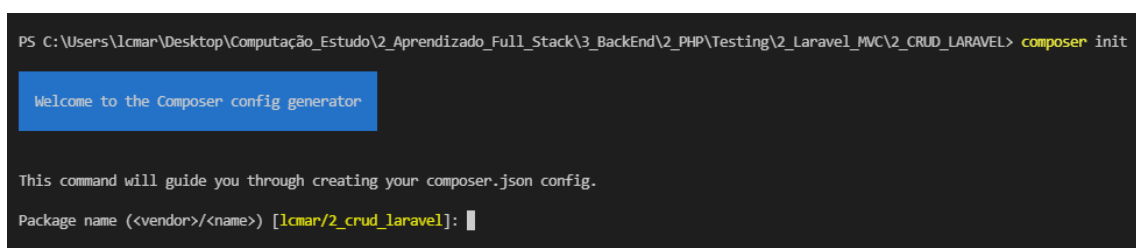
Existem PSRs que tratam de diversos temas como: estilo de código, autoload (a famosa PSR-4), cache, log, HTTP e outros. É possível ver a [lista de PSRs](#) e seus estados no site do PHP-FIG.

2. COMO FUNCIONA O COMPOSER

2.1. INICIALIZANDO O COMPOSER NO PROJETO

O composer, então, foi instalado na máquina, tem uma variável de ambiente, e é uma ferramenta utilizada, por padrão, localmente, em projetos.

Agora, existindo um projeto, para inicializar o composer o local dele deve ser acessado via linha de comando, e o comando “**composer init**” deve ser escrito e executado. Esse comando dá início a uma espécie de formulário para as definições do composer.json.



```
PS C:\Users\lcmar\Desktop\Computação_Estudo\2_Aprendizado_Full_Stack\3_BackEnd\2_PHP\Testing\2_Laravel_MVC\2_CRUD_LARAVEL> composer init

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [lcmar/2_crud_laravel]:
```

O exemplo acima foi realizado no terminal integrado do Visual Studio Code. A pasta foi acessada no terminal, e o comando foi executado. Os campos que exigem dados são para preenchimento aqueles básicos descritos no tópico sobre a esquematização do manifesto.

Quando o manifesto for concluído, alguns arquivos e pastas serão gerados, como ele mesmo – o composer.json -, o composer.lock e a pasta vendor, que é utilizada para armazenar as dependências externas instaladas, bem como uma pasta chamada “composer” que possui os scripts da ferramenta.

2.2. INSTALANDO PACOTES EXTERNOS

Para instalar pacotes externos, que são classes criadas por outros desenvolvedores para desempenhar determinadas funções, **existem duas formas**, e que inclusive já foram comentadas.

A primeira forma é com o comando “composer require [package]”, que instala o pacote e edita o manifesto automaticamente.

A segunda forma é escrevendo manualmente o nome e a versão do pacote no manifesto, e em seguida executando comando “composer update” para que o manifesto seja analisado, a nova dependência detectada e instalada segundo as especificações definidas.

2.3. MAPEANDO AS CLASSES INTERNAS: NAMESPACE E DUMP-AUTOLOAD

Para utilizar classes criadas no projeto, localmente, com o autoload, elas devem ser mapeadas. Pois bem: o que permite esse mapeamento, e que o autoload utiliza como fator de procura, são os namespaces.

No manifesto, no campo “autoload”, seguindo a especificação PSR-4, são declarados namespaces para locais existentes no projeto, e que são como nomes alternativos. Vamos considerar o exemplo abaixo:

```
“autoload”: { “psr-4”: { “Classes\\”: “Classes/” } }
```

A primeira consideração importante é quanto a sintaxe do namespace no manifesto. Ele é uma chave que possui um valor. Seu nome deve ser “[nome]\\”, e seu valor correspondente um caminho do projeto. Neste caso, entende-se que o namespace “Classes” será um nome, ou apelido, para o caminho “Classes/” existente na raiz do projeto.

O próximo passo, agora existindo esse namespace, é marcar as classes com ele para que sejam consideradas pelo autoload. Pois bem: se o arquivo da classe existir em “Classes/” seu namespace será, sem aspas, claro, “namespace Classes”. Mas, e isso é muito importante, **se** o arquivo ainda estiver em uma subpasta, como “Classes/Exemplo/”, seu namespace será “namespace Classes\Exemplo”.

A marcação com o namespace, então, das classes internas, é uma marcação do local onde o autoload deve procurar. Se o namespace “Classes” é um apelido para a pasta “Classes/”, as classes existentes em subpastas dessa pasta não poderão ser marcadas apenas com o apelido “Classes”, porque não estarão exatamente em “Classes/”, mas sim em níveis acima.

Após a definição do namespace, e marcação das classes da forma correta, deve ser utilizado o comando “**composer dump-autoload**”.

Ele, diferente do “update”, que verifica a declaração das dependências externas, lida com o mecanismo “classmap” do composer, que é o mapeamento das classes internas. O que o dump-autoload faz, então, é **atualizar o mapeamento das classes internas**.

2.4. O AUTOLOAD E USO DAS CLASSES

Para utilizar os recursos mapeados e as dependências em um arquivo, o primeiro passo é realizar o require do autoload, desta forma: require(“vendor/autoload.php”). Isso abre as portas para que os recursos sejam utilizados um a um.

2.4.1. UTILIZANDO AS CLASSES INTERNAS MAPEADAS

O segundo passo, que consiste em de fato fazer uso desses recursos, requer que se utilize o termo reservado “use”. Para classes internas, a sintaxe é a seguinte:

use [namespace da classe]\[nome da classe]

Desta forma, a instância da classe se torna possível a partir do seu nome original. Mas, ainda é possível utilizar um apelido para o nome da classe. Com apelido a sintaxe é a seguinte:

use [namespace da classe]\[nome da classe] as [apelido]

Desta forma, a instância da classe se torna possível a partir do seu apelido. Por exemplo, ao invés de instanciar a classe “Pessoa” como \$obj = new Pessoa(), pode ser \$obj = new Example(), se for definido o apelido “Example”.

Também é possível realizar a instância da classe a partir do seu namespace\[nome original], diretamente, dispensando o uso do termo reservado “use”. Por exemplo:

[namespace da classe]\[nome da classe]()

2.4.2. UTILIZANDO AS DEPENDÊNCIAS EXTERNAS INSTALADAS

O funcionamento do uso dos pacotes externos é exatamente o mesmo do uso das classes internas, a diferença é que nós conhecemos os namespaces das classes que criamos, mas não os das classes externas instaladas.

“Devo escrever use ...o quê? Qual o namespace dessa classe externa?”. Isso não é um problema. Na verdade, as documentações dos pacotes sempre informam o usuário sobre como utilizar as classes. Clique [aqui](#) para ver por si mesmo o exemplo da imagem abaixo.

Usage

Generate a slug:

```
use Cocur\Slugify\Slugify;  
  
$slugify = new Slugify();  
echo $slugify->slugify('Hello World!'); // hello-world
```