PHP Code Generator Documentation

Release

Thomas Gossmann

Jul 01, 2017

Contents

This is a code generator for php code.

CHAPTER 1

Quickstart

1. Install: `composer require gossi/php-code-generator`
2. You need a *Model*
3. You need a *Generator*
4. Generate the code contained in the model

Contents:

Installation

Install via Composer:

```
{
    "require": {
        "gossi/php-code-generator": "~0"
    }
}
```

or via CLI:

```
composer require 'gossi/php-code-generator'
```

Getting Started

There are two things you need to generate code.

1. **A *Model* that contains the code structure**
    - PhpClass
    - PhpInterface

- PhpTrait
- PhpFunction

2. A *Generator*
    - CodeGenerator
    - CodeFileGenerator

You can create these models and push all the data using a fluent API or read from existing code through reflection. Here are two examples for each of those.

Generate Code

1. Simple:

```php
<?php
use gossi\codegen\generator\CodeGenerator;
use gossi\codegen\model\PhpClass;
use gossi\codegen\model\PhpMethod;
use gossi\codegen\model\PhpParameter;

$class = new PhpClass();
$class
    ->setQualifiedName('my\\cool\\Tool')
    ->setMethod(PhpMethod::create('__construct')
        ->addParameter(PhpParameter::create('target')
            ->setType('string')
            ->setDescription('Creates my Tool')
        )
    )
;

$generator = new CodeGenerator();
$code = $generator->generate($class);
```

will generate:

```php
<?php
namespace my\cool;

class Tool {

    /**
     *
     * @param $target string Creates my Tool
     */
    public function __construct($target) {
    }
}
```

2. From File:

```php
<?php
use gossi\codegen\generator\CodeGenerator;
use gossi\codegen\model\PhpClass;

$class = PhpClass::fromFile('path/to/class.php');
```

```
$generator = new CodeGenerator();
$code = $generator->generate($class);
```

3. From Reflection:

```
    <?php
    use gossi\codegen\generator\CodeGenerator;
    use gossi\codegen\model\PhpClass;

$reflection = new \ReflectionClass('MyClass');
    $class = PhpClass::fromReflection($reflection->getFileName());

    $generator = new CodeGenerator();
    $code = $generator->generate($class);
```

Model

A model is a representation of your code, that you either read or create.

Model Overview

There are different types of models available which are explained in this section.

Structured Models

Structured models are composites and can contain other models, these are:

- `PhpClass`
- `PhpTrait`
- `PhpInterface`

Generateable Models

There is only a couple of models available which can be passed to a generator. These are the mentioned structured models + `PhpFunction`.

Part Models

Structured models can be composed of various members. And functions and methods can itself contain zero to many parameters. All parts are:

- `PhpConstant`
- `PhpProperty`
- `PhpMethod`
- `PhpParameter`

Name vs. Namespace vs. Qualified Name ?

There can be a little struggle about the different names, which are *name*, *namespace* and *qualified name*. Every model has a name and generateable models additionally have a namespace and qualified name. The qualified name is a combination of namespace + name. Here is an overview:

Name	Tool
Namespace	my\cool
Qualified Name	my\cool\Tool

Create Models programmatically

You can create models with the provided fluent API. The functionality is demonstrated for a `PhpClass` but also works accordingly for all the other generateable models.

Create your first Class

Let's start with a simple example:

```
<?php
use gossi\codegen\model\PhpClass;

$class = new PhpClass();
$class->setQualifiedName('my\\cool\\Tool');
```

which will generate an empty:

```
<?php
namespace my\cool;

class Tool {

}
```

Adding a Constructor

It's better to have a constructor, so we add one:

```
<?php
use gossi\codegen\model\PhpClass;
use gossi\codegen\model\PhpMethod;
use gossi\codegen\model\PhpParameter;

$class = new PhpClass('my\\cool\\Tool');
$class
    ->setMethod(PhpMethod::create('__construct')
        ->addParameter(PhpParameter::create('target')
            ->setType('string')
            ->setDescription('Creates my Tool')
        )
    )
;
```

you can see the fluent API in action and the snippet above will generate:

```php
<?php
namespace my\cool;

class Tool {

    /**
     *
     * @param $target string Creates my Tool
     */
    public function __construct($target) {
    }
}
```

Adding members

We've just learned how to pass a blank method, the constructor to the class. We can also add *properties*, *constants* and of course *methods*. Let's do so:

```php
<?php
use gossi\codegen\model\PhpClass;
use gossi\codegen\model\PhpMethod;
use gossi\codegen\model\PhpParameter;
use gossi\codegen\model\PhpProperty;
use gossi\codegen\model\PhpConstant;

$class = PhpClass::create('my\\cool\\Tool')
    ->setMethod(PhpMethod::create('setDriver')
        ->addParameter(PhpParameter::create('driver')
            ->setType('string')
        )
        ->setBody('$this->driver = $driver');
    )
    ->setProperty(PhpProperty::create('driver')
        ->setVisibility('private')
        ->setType('string')
    )
    ->setConstant(new PhpConstant('FOO', 'bar'))
;
```

will generate:

```php
<?php
namespace my\cool;

class Tool {

    /**
     */
    const FOO = 'bar';

    /**
     * @var string
     */
    private $driver;

    /**
```

```
     *
     * @param $driver string
     */
    public function setDriver($driver) {
        $this->driver = $driver;
    }
}
```

Declare use statements

When you put code inside a method there can be a reference to a class or interface, where you normally put the qualified name into a use statement. So here is how you do it:

```
<?php
use gossi\codegen\model\PhpClass;
use gossi\codegen\model\PhpMethod;

$class = new PhpClass();
$class
    ->setName('Tool')
    ->setNamespace('my\\cool')
    ->setMethod(PhpMethod::create('__construct')
        ->setBody('$request = Request::createFromGlobals();')
    )
    ->declareUse('Symfony\\Component\\HttpFoundation\\Request')
;
```

which will create:

```
<?php
namespace my\cool;

use Symfony\Component\HttpFoundation\Request;

class Tool {

    /**
     */
    public function __construct() {
        $request = Request::createFromGlobals();
    }
}
```

Much, much more

The API has a lot more to offer and has almost full support for what you would expect to manipulate on each model, of course everything is fluent API.

Create from existing Models

You can also read a model from existing code. Reading from a file is probably the best option here. It will parse the file and fill up the model. Alternatively if you do have the class already loaded you can use reflection to load the model.

From File

Reading from a file is the simplest way to read existing code, just like this:

```
<?php
use gossi\codegen\model\PhpClass;

$class = PhpClass::fromFile('path/to/MyClass.php');
```

Through Reflection

If you already have your class loaded, then you can use reflection to load your code:

```
<?php
use gossi\codegen\model\PhpClass;

$reflection = new \ReflectionClass('MyClass');
$class = PhpClass::fromReflection($reflection->getFileName());
```

Also reflection is nice, there is a catch to it. You must make sure `MyClass` is loaded. Also all the requirements (use statements, etc.) are loaded as well, anyway you will get an error when initializing the the reflection object.

Understanding Values

The models `PhpConstant`, `PhpParameter` and `PhpProperty` support values; all of them implement the `ValueInterface`. Though, there is a difference between values and expressions. Values refer to language primitives (`string`, `int`, `float`, `bool` and `null`). Additionally you can set a `PhpConstant` as value (the lib understands this as a library primitive ;-). If you want more complex control over the output, you can set an expression instead, which will be generated as is.

Some Examples:

```
<?php
PhpProperty::create('foo')->setValue('hello world.');
// $foo = 'hello world.';

PhpProperty::create('foo')->setValue(300);
// $foo = 300;

PhpProperty::create('foo')->setValue(3.14);
// $foo = 3.14;

PhpProperty::create('foo')->setValue(false);
// $foo = false;

PhpProperty::create('foo')->setValue(null);
// $foo = null;

PhpProperty::create('foo')->setValue(PhpConstant::create('BAR'));
// $foo = BAR;

PhpProperty::create('foo')->setExpression('self::MY_CONST');
// $foo = self::MY_CONST;

PhpProperty::create('foo')->setExpression("['my' => 'array']");
// $foo = ['my' => 'array'];
```

For retrieving values there is a `hasValue()` method which returns `true` whether there is a value or an expression present. To be sure what is present there is also an `isExpression()` method which you can use as a second check:

```
<?php

if ($prop->hasValue()) {
    if ($prop->isExpression()) {
        // do something with an expression
    } else {
        // do something with a value
    }
}
```

Generator

The package ships with two generators, which are configurable through an associative array as constructor parameter. Alternatively if you have a project that uses the same configuration over and over again, extend the respective config object and pass it instead of the configuration array.

```
<?php
use gossi\codegen\generator\CodeGenerator;

// a) new code generator with options passed as array
$generator = new CodeGenerator([
  'generateDocblock' => false,
  ...
]);

// b) new code generator with options passed as object
$generator = new CodeGenerator(new MyCodeGenerationConfig());
```

CodeGenerator

Generates code for a given model. Additionally (and by default), it will generate docblocks for all contained classes, methods, interfaces, etc. you have prior to generating the code.

- Class: `gossi\codegen\generator\CodeGenerator`
- Config: `gossi\codegen\config\CodeGeneratorConfig`
- Options:

Key	Type	Default Value	Description
generate-Docblock	boolean	true	enables docblock generation prior to code generation
generateEmpty-Docblock	boolean	true	allows generation of empty docblocks
generateScalar-TypeHints	boolean	false	generates scalar type hints, e.g. in method/function parameters (PHP 7)
generateReturn-TypeHints	boolean	false	generates scalar type hints for return values (PHP 7)
enableSorting	boolean	true	Enables sorting
useState-mentSorting	boolean|string|Closure|Comparator	default	Sorting mechanism for use statements
constantSorting	boolean|string|Closure|Comparator	default	Sorting mechanism for constants
propertySorting	boolean|string|Closure|Comparator	default	Sorting mechanism for properties
methodSorting	boolean|string|Closure|Comparator	default	Sorting mechanism for methods

Note: when `generateDocblock` is set to `false` then `generateEmptyDocblock` is `false` as well.

Note 2: For sorting ...

- ... a string will used to find a comparator with that name (at the moment there is only default).
- ... with a boolean you can disable sorting for a particular member
- ... you can pass in your own `\Closure` for comparison
- ... you can pass in a Comparator for comparison

- Example:

```
<?php
use gossi\codegen\generator\CodeGenerator;

// will set every option to true, because of the defaults
$generator = new CodeGenerator([
  'generateScalarTypeHints' => true,
  'generateReturnTypeHints' => true
]);
$code = $generator->generate($myClass);
```

CodeFileGenerator

Generates a complete php file with the given model inside. Especially useful when creating PSR-4 compliant code, which you are about to dump into a file. It extends the `CodeGenerator` and as such inherits all its benefits.

- Class: `gossi\codegen\generator\CodeFileGenerator`
- Config: `gossi\codegen\config\CodeFileGeneratorConfig`
- Options: Same options as `CodeGenerator` plus:

Key	Type	Default Value	Description
headerComment	null|string|Docblock	null	A comment, that will be put after the `<?php` statement
headerDocblock	null|string|Docblock	null	A docblock that will be positioned after the possible header comment
blankLineAtEnd	boolean	true	Places an empty line at the end of the generated file
declareStrictTypes	boolean	false	Whether or not a `declare(strict_types=1);` is placed at the top of the file (PHP 7)

Note: `declareStrictTypes` sets `generateScalarTypeHints` and `generateReturnTypeHints` to `true`.

- Example:

```
<?php
use gossi\codegen\generator\CodeFileGenerator;

$generator = new CodeGenerator([
  'headerComment' => 'This will be placed at the top, woo',
  'headerDocblock' => 'Full documentation mode confirmed!',
  'declareStrictTypes' => true
]);
$code = $generator->generate($myClass);
```

Best Practices

The code generator was written with some thoughts in mind. See for yourself, if they are useful for you, too.

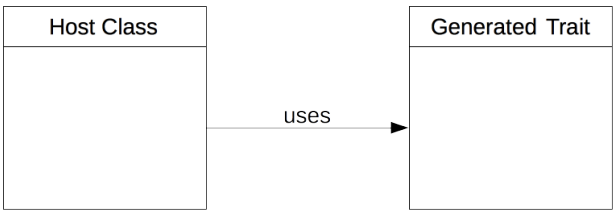
Template system for Code Bodies

It is useful to use some kind of template system to load the contents for your bodies. The template system can also be used to replace variables in the templates.

Hack in Traits

Let's assume you generate a php class. This class will be used in your desired framework as it serves a specific purpose in there. It possible needs to fulfill an interface or some abstract methods and your generated code will also take care of this - wonderful. Now imagine the programmer wants to change the code your code generation tools created. Once you run the code generation tools again his changes probably got overwritten, which would be bad.

Here is the trick: First we declare the generated class as "host" class:



Your generated code will target the trait, where you can savely overwrite code. However, you must make sure the trait will be used from the host class and also generate the host class, if it doesn't exist. So here are the steps following this paradigm:

1. Create the trait
2. Check if the host class exists
   (a) if it exists, load it
   (b) if not, create it
3. Add the trait to the host class
4. Generate the host class code

That way, the host class will be user-land code and the developer can write his own code there. The code generation tools will keep that code intact, so it won't be destroyed when code generation tools run again. If you want to give the programmer more freedom offer him hook methods in the host class, that - if he wants to - can overwrite with his own logic.

Format in Post-Processing

After generating code is finished, it can happen that (especially) bodies are formatted ugly. Thus just run the suggested code formatter after generating the code. Can be found on github gossi/php-code-formatter.

API

API is available at https://gossi.github.io/php-code-generator/api/master