Dogmatist

Release 1.0

May 04, 2015

Contents

Dogmatist is a simple framework for specifying the structure of one or more objects and arrays and the relations between those object and arrays.

Using this description Dogmatist will allow you to generate specific instances, called samples.

CHAPTER 1

Getting started

This document contains a quick introduction on how to get started with some very basic code examples.

1.1 Requirements

Dogmatist requires PHP 5.5 or higher. Generally for installing it you should be using Composer.

1.2 Installing

1.2.1 Using Composer

To install Dogmatist as a composer dependency of your project, use:

```
composer require bravesheep/dogmatist=dev-master
```

To install Dogmatist as a development dependency, which is what you will most likely be using Dogmatist for, you can use this command:

```
composer require --dev bravesheep/dogmatist=dev-master
```

Note that this installs the master branch variant of Dogmatist. If you prefer to use a stable release, you can choose to omit the `=dev-master` part of the command which should ensure that you get the latest stable release.

1.2.2 Cloning from git

You can also clone and install the project from git using:

```
git clone https://github.com/bravesheep/dogmatist.git
cd dogmatist
composer install
```

1.3 Constructing dogmatist

In order to work with Dogmatist, you will need an instance of the base Dogmatist generator class. You can either manually construct an instance, or (preferred) use the `Factory` class provided to create an instance. When you're

using composer, also make sure you have included the autoloader, so all classes get automatically loaded from that point on:

```
<?php

require "vendor/autoload.php";

$dogmatist = \Bravesheep\Dogmatist\Factory::create();
```

Note: Future code fragments will omit things like the PHP open tag and `require` statement. In general `$dogmatist` will be an instance of `Dogmatist` created using this factory method.

The `Factory::create()` method may be provided with an optional set of arguments. Three things can be provided by the programmer:

1. An instance of `Faker\Generator` or a string specifying the language of the generator which should be constructed.
2. An instance of `Bravesheep\Dogmatist\Guesser\GuesserInterface`. A guesser can be used to automatically determine what should be generated.
3. An instance of `Symfony\Component\PropertyAccess\PropertyAccessorInterface` for setting properties. One will be constructed for you if you don't provide any, however this may be used to provide an already existing instance.

So a Dogmatist instance that would be using French providers, could be constructed using this for example:

```
$dogmatist = \Bravesheep\Dogmatist\Factory::create('fr_FR');
```

Now that we have a dogmatist example we can start generating builders and samples from them:

```
$dogmatist = \Bravesheep\Dogmatist\Factory::create();
$dogmatist
    ->create('object')
        ->fake('name', 'name')
        ->value('count', 1)
        ->relation('address', 'object')
            ->fake('address', 'streetAddress')
            ->fake('city', 'city')
        ->done()
        ->multiple('address', 1, 3)
        ->save('person', 5)
    ->done()
;
$persons = $dogmatist->freshSamples('person', 10);
```

Now that you've seen an example take a look at the rest of the documentation to learn about the interface that Dogmatist provides for generating objects and entities.

CHAPTER 2

Describing objects

In Dogmatist, you describe objects using a simple fluent syntax. This descriptive method results in a `Builder` object being created. Feeding this `Builder` into the sampler will allow Dogmatist to create an instance of that object.

2.1 Starting a description

To start an object description, use the `create()` method on the `Dogmatist` instance, like so for creating a description for an array:

```
// $dogmatist is an instance of ''Dogmatist''
$builder = $dogmatist->create('array');
```

The first (and only) argument of the `create()` method specifies the type of object that should be created. You can specify the type as follows:

- Using `array`. This will not create an object, but a native PHP array instead.
- Using `object` (or `stdClass`): this will cast the array as previously specified to an object, and thus will return an instance of `stdClass`.
- Using a fully namespaced class name. Note that the class name needs to be loadable using some sort of autoloader and that it should be possible to create an instance. Dogmatist does not work with interfaces, traits or abstract classes, only concrete (or final) classes can be used.

Note: This documentation often uses the PHP 5.5+ syntax of `::class`. This syntax allows you to get the full namespaced classname when all you have is an imported symbol name of some class. This prevents having to write down the class name as a string, which is error-prone and does not allow for easy refactoring.

Say we have a Doctrine2 ORM entity in our project called `Acme\Entity\Person` then we might for example use this to create a builder for that class:

```
use Acme\Entity\Person;

$builder = $dogmatist->create(Person::class);
```

2.2 Builder options

A Builder instance has some methods to manipulate and retrieve its state:

isStrict() When a Builder is set to strict mode, then generating a field which doesn't exist in the target class, or cannot be reached because it is protected or private and has not accessor function, then the Builder will trigger an error. In non-strict mode, the Builder is allowed to use reflection to write any property to an object.

setStrict(strictness) Set the strictness (a boolean `true` or `false`). Note that this also sets strictness of the child-builders and constructor builder.

setParent(parent) Sets the parent of this Builder. You don't need to call this method directly, as the Builder will automatically set up parent-child relations.

constructor() Used to retrieve the constructor Builder. You can read more about the constructor builder in the section below.

hasConstructor() Returns whether or not the Builder actually has a constructor.

get(field) Retrieves a field with a specific name. More about setting up fields is described in the section below.

done() Returns the parent Builder for this one, indicating that the builder is complete. If a Builder has no parent, then the associated instance of Dogmatist is returned. This allows you to chain another call to `create()` to start

save(name[, count]) Saves the builder in dogmatist, to be used for linking or for generating samples directly from dogmatist using a named descriptor. The count parameter is used to indicate the maximum number of samples that should be generated. If this count is set to any number equal to or less than zero then an unlimited number of samples may be generated.

getFields() Retrieves the fields that have been defined for this builder. You can read more about fields in the section below.

getType() Retrieves the type name for which this Builder will generate samples.

onCreate(callback) Provide a callback function that will be called using a newly created sample. Note that you can call this method multiple times with different callback functions which will all be called when a new samples is created.

getListeners() Retrieve a list of callbacks that should be called when a new samples is created.

copy([type]) Creates a deep copy of the builder, if the specified type is given, the type of the builder will be adjusted to the new type.

setType(type) Set a new type for the builder. The builder is then configured for constructing objects of that type. Note that you still have to make sure all fields are available.

2.3 Describing fields

Once a Builder instance is retrieved using the create method you have several methods to describe the fields contained in that class. The methods available are described below:

none(field) A field described using this method will ensure that the field will never be touched when sampling. By default all fields in an object won't be touched. So this method is mainly used to remove fields which have previously been added.

fake(field, type[, options]) The fake method is used to describe a field in your object which should get some random value generated by Faker. The type should be a type that is available in the `Faker\Generator` instance that Dogmatist has. You can use one of the default providers Faker provides, such as `randomNumber` or `country`, or you can create your own Providers and register them in the Generator instance to have more ways of specifying field values.

Note that you can specify an array of options that should be passed on to the generator. If you were to call a faker method directly these would normally be the arguments entered here.

select(field, options) The select field allows you to specify that a random value from a list of predetermined values should be picked. Note that this is equivalent to using `fake($field, 'randomElement', [$options])`, however since select fields may occur quite often (such as in the case of Male/Female or true/false) they have been given a special descriptive function.

value(field, value) Set a predetermined value for a specific field. This means that all samples of the object will always have the same value for a specific field. You can for example use this when describing something like an active user, where that active flag is indicated using a boolean which should always be true for active users.

link(field, value) You can link one builder to another builder using this method. For a better description you should take a look at the section on saved builders. Note that you can also specify an array of linked builders, in which case one will be selected randomly for each sample created.

relation(field, type) This is the most complicated of all the available functions. The relation type of description allows you to describe a sub-builder for that specific property. For example take some User class which contains an Address.

callback(field, callback) Calls the function callback with as the first arguments an array of all fields generated up to this point, and as a second argument the associated instance of Dogmatist. This callback function should return a value to be used at that position. Note that you can set a callback field to multiple (as seen below), in that case the callback will be called multiple times. Also note that you don't have access to any fields described after having described this one, as they still have to be generated.

The previous methods all allow you to describe the type of the field. Every field can either generate an array of values or just a single value. By default all fields will be singular, but using the following two methods you can change this behavior:

singe(field) Sets a field to only produce a single value when a sample is generated.

multiple(field[, min, max]) Sets a field to produce at least min and at most max values. These values are combined as an array. If an add method is provided however, the results will be inserted one item at a time into the field. If no such method exists then the value will be inserted directly.

Every field can be set to generate only unique values. By the default the sampler will try a limited number of times to try and generate a unique value. If that proves to be impossible within that limit, the sampler will fail to generate a new value. To mark a field for uniqueness, you can use the following method:

unique(field[, uniqueness]) When called will mark a field for uniqueness if called with one argument, otherwise a boolean may be provided as the second argument indicating the uniqueness of the field.

For the `single`, `multiple` and `unique` calls you will often want to apply these functions to the field you have just created. In order to help you with this use case you can use these methods in camel-cased variants prefixed with `with` to access the previously created field:

withSingle() Mark the previously accessed field as singular.

withMultiple([min, max]) Mark the previously accessed field as multiple with the specified min and max.

withUnique([uniqueness] Mark the field as being unique.

2.4 Describing the constructor

Using the `constructor()` method, you can create a description for constructing the object. Inside this `ConstructorBuilder` you can mostly use the same methods as with a normal `Builder` object. However, you cannot save a constructor, nor can you add a constructor to a constructor recursively, and finally you cannot add listeners using the `onCreate()` method.

When describing the constructor you can choose one of two methods:

Named Describe using the names of the constructor arguments. This is done using the same methods as describing fields in a normal Builder.

Positional Using positional arguments with the `arg*` methods: `argFake()`, `argSelect()`, `argValue()`, `argLink()`, `argRelation()` and `argCallback()`. These methods have the same signature as their named counterparts, except that you can leave out the name of the field.

To determine if a constructor is using named or positional arguments, you can use the `isPositional()` method on the constructor builder.

Note: You cannot mix positional and named arguments in the constructor. If you try to do this, you will get a `BuilderException`.

2.5 Relating back to the parent object

When creating a sub-builder using the `relation()` method, you can specify a field that should be updated with the parent object. To do this, use these special builder methods:

`linkParent(field)` Will insert the parent object in the specified field.

`hasLinkWithParent()` Returns whether or not this builder wants to create a link to the parent.

`getLinkParent()` Retrieves the field into which the parent should be inserted.

Note: Inserting the parent also works with `stdClass` objects and arrays.

CHAPTER 3

Sampling

Once you have described your object you can use the sampler to create samples for your object. To create samples you can use one of four methods inside the `Dogmatist` instance:

`sample(builder)` When builder is a name, a sample from the named builder is created. If no builder can be found then a `NoSuchIndexException` is thrown. When the named builder has a limited set of samples, then one of these samples is returned. If builder is a named builder that is allowed to generate unlimited samples, or if builder is an instance of `Builder`, then a fresh sample is generated.

`samples(builder, count)` This method is similar in behavior to the `sample()` method, however this method returns a provided number of samples. Note that if a builder only has a limited set of samples it is allowed to generate and you ask for more, then the sampler will throw a `SampleException` indicating that not enough samples are available. For `Builder` instances or named builders that are allowed to generate unlimited samples, a sample will always be generated as long as the Builder is valid for the type it describes.

`freshSample(builder)` This method will always generate a fresh sample, even for builders which are only allowed to generate a limited set of samples.

`freshSamples(builder, count)` Similar to `freshSample()`, this will always generate fresh samples.

Note: When a sample is fresh, this does not mean that it is unique from all previously generated samples when looked at the field level. However the sampler will have at least run it's course once more and in the case of objects, no freshly generated sample will ever be exactly equal (===) to any of the previously generated samples.

```
$builder = $dogmatist->create('object')->fake('num', 'randomNumber');
$sample = $dogmatist->sample($builder);
var_dump($sample);
```

CHAPTER 4

Guessers

In Dogmatist, a guesser is a class that automatically tries to determine the fields used.

CHAPTER 5

Examples

In this chapter are some example usages of Dogmatist, the Builder and how to create samples.

5.1 Basic example

```
// start by creating a dogmatist instance
$dogmatist = \Bravesheep\Dogmatist\Factory::create();

// let's create two main builders: one for Person objects and one for
// Post object. The Person object gets its own Address builder which will
// generate specific instances per person
$dogmatist
    ->create(Person::class)
        ->fake('name', 'name')
        ->fake('birthday', 'datetimeBetween', ['-100 years', '-15 years'])
        ->fake('company', 'company')
        ->relation('addresses', Address::class)
            ->fake('address', 'streetAddress')
            ->fake('postcode', 'postcode')
            ->fake('city', 'city')
            ->fake('country', 'country')
        ->done()
        ->multiple('addresses', 1, 5)
        ->select('gender', ['male', 'female'])
        ->save('person', 10)
    ->done()
    ->create(Post::class)
        ->link('poster', 'person')
        ->fake('title', 'sentence')
        ->fake('text', 'text', [5000])
        ->value('visible', true)
        ->save('post')
    ->done()
;

// gets us one of the 10 persons that could be created
$person = $dogmatist->sample('person');

// gets us a random new post
$post = $dogmatist->sample('post');
```

```
try {
    // will throw an exception
    $people = $dogmatist->samples('person', 20);
} catch (SampleException $e) {
    print $e->getMessage();
}

// this will generate an array of 20 Person objects
$people = $dogmatist->freshSamples('person', 20);
```

5.2 Events

```
// start by creating a dogmatist instance
$dogmatist = \Bravesheep\Dogmatist\Factory::create();

// here we add a callback to append the slug, which has to base its value
// on another faked value in the object
$builder = $dogmatist
    ->create(Post::class)
        ->fake('title', 'person')
        ->fake('content', 'text', [5000])
        ->onCreate(function (&$value) {
            $obj->setSlug(Slugifier::slugize($obj->getTitle()));
        });

$sample = $dogmatist->sample($builder);

// so now we have a slug
print $sample->getSlug();
```