Jane Documentation Documentation

Release 4.0.0

Joel Wurtz

Jun 12, 2019

Contents

Jane regroups different tools that aim to generated high quality PHP code respecting common and advanced PSR.

- JsonSchema
- OpenAPI

CHAPTER 1

Internal

This documentation describes how all Jane libraries work to generate the code. It is mainly oriented for people wanting to contribute to this library.

This library is based on 3 different steps:

1.1 Guessing

First step is to guess a set of metadata given a specification (JsonSchema or OpenAPI at the time of writing this). To do so, it will read the specification, transform it into objects and pass it to guessers implementing one of the `GuesserInterface`.

Each guesser tell if it supports the current specification and returns metadata. Occasionally, it will try to guess sub objects of the specification.

1.2 Analyzing

Once all metadata are guessed, they are passed to a set of generators implementing the `GeneratorInterface` given a `Context`.

Then, each generator will analyze the metadata and create PHP code by using the PHP Parser Library. Using the library improves the flexibility to create complex code, as using a template generator solution.

`Context` provides a lots of functions to generate code, like using unique variable name in a scope or adding generated file.

1.3 Generation

When the code is ready, the `Context` is read to generate PHP files and optionally format it with PHP CS Fixer if available.

CHAPTER 2

Backwards compatibility

Backwards compatiblity is an important topic. Those libraries follow Semver, so backwards compatibility will only break between major versions. This library may use deprecations notices to inform you of the change, but it's a low probability, you should always check the CHANGELOG when switching to a new major version.

2.1 JsonSchema and OpenAPI

Those libraries generate code and should not be used in runtime. Also, there is no need to extends or use this code in another libraries. The only thing used, is the command line.

So there is no BC promise on those libraries, you can consider that everything is internal. The only BC promise is about the command line, and the generated code.

2.2 Generated Code

Code generated fall under our BC Promise, but only the public and protected API of the generated code. When a method of a class is generated, its signature will not change with minor release, but it's implementation may change, however a private method can have its signature updated. Behavior of the implementation should not change between minor releases unless behavior is buggy.

No class will be removed between minor versions, but there can be new classes added.

2.3 Runtime Libraries

JsonSchema Runtime and OpenAPI Runtime libraries have a standard BC Promise.

CHAPTER 3

Json Schema

Jane JsonSchema is a library to generate models and serializers in PHP from a JSON Schema draft v4.

3.1 Installation

Add this library with composer as a `dev` dependency:

```
composer require --dev jane-php/json-schema "^4.0"
```

This library contains a lot of dependencies to be able to generate code which are not needed on runtime. However, the generated code depends on other libraries and a few classes that are available through the runtime package. It is highly recommended to add the runtime dependency as a requirement through composer:

```
composer require jane-php/json-schema-runtime "^4.0"
```

By default, generated code is not formatted, to make it compliant to PSR2 standard and others format norms, you can add the PHP CS Fixer library to your dev dependencies (and it makes it easier to debug!):

```
composer require --dev friendsofphp/php-cs-fixer "^2.7.3"
```

CHAPTER 4

Generating a Model

This library provided a PHP console application to generate the Model, you can use it by executing the following command at the root of your project:

```
php vendor/bin/jane generate
```

This command will try to read a config file named `.jane` located on the current working directory. However, you can name it as you like and use the `--config-file` option to specify its location and name:

```
php vendor/bin/jane generate --config-file=jane-configuration.php
```

Note: No others options can be passed to this command. Having a config file ensure that a team working on the project always use the same set of parameters and, when it changes, give vision of the new option used to generate the code.

4.1 Configuration file

The configuration file consists of a simple PHP script returning an array:

```php
<?php

return [
    'json-schema-file' => __DIR__ . '/json-schema.json',
    'root-class' => 'MyModel',
    'namespace' => 'Vendor\Library\Generated',
    'directory' => __DIR__ . '/generated',
];
```

This example shows the minimum configuration required to generate a Model:

- `json-schema-file`: Specify the location of your json schema file, it can be a local file or a remote one `https://my.domain.com/my-schema.json`

- `root-class`: The root class of the root object defined in your json schema, if there is no property on the root object it will not be used
- `namespace`: Root namespace of all of your generated code
- `directory`: Directory where the code will be generated at

Given this configuration you will need to add the following configuration to composer, in order to load the generated files:

```
"autoload": {
    "psr-4": {
        "Vendor\\Library\\Generated\\": "generated/"
    }
}
```

4.2 Options

Other options are available to customize the generated code:

- `reference`: A boolean which indicate to add the support for JSON Reference into the generated code.
- `date-format`: A date format to specify how the generated code should encode and decode `\DateTime` object to string
- `use-fixer`: A boolean which indicate if we make a first cs-fix after code generation
- `fixer-config-file`: A string to specify where to find the custom configuration for the cs-fixer after code generation
- `use-cacheable-supports-method`: A boolean which indicate if we use `CacheableSupportsMethodInterface` interface to improve caching performances when used with Symfony Serializer.

4.3 Multi schemas

Jane JsonSchema can also generate multiple schemas at the same time with different namespaces and directories, allowing to handle JSON References on others schemas.

See *Multi schemas generation* for more information

CHAPTER 5

Using a generated Model

This library generates basics P.O.P.O. objects (Plain Old PHP Objects) with a bunch of setters / getters. It also generates all normalizers to handle denormalization from a json string, and normalization.

All normalizers respect the `Symfony\Component\Serializer\Normalizer\NormalizerInterface` and `Symfony\Component\Serializer\Normalizer\DenormalizerInterface` from the Symfony Serializer Component.

It also generates a `NormalizerFactory` class having a static function `create` returning an array of all normalizers.

Given this configuration:

```php
<?php

return [
    'json-schema-file' => __DIR__ . '/json-schema.json',
    'root-class' => 'MyModel',
    'namespace' => 'Vendor\Library\Generated',
    'directory' => __DIR__ . '/generated',
];
```

You will have to do this:

```php
<?php

$normalizers = Vendor\Library\Generated\Normalizer\NormalizerFactory::create();
$encoders = [new Symfony\Component\Serializer\Encoder\JsonEncoder(
    new Symfony\Component\Serializer\Encoder\JsonEncode(JSON_UNESCAPED_SLASHES),
    new Symfony\Component\Serializer\Encoder\JsonDecode(false))
];

$serializer = new Symfony\Component\Serializer\Serializer($normalizers, $encoders);
$serializer->deserialize('{...}');
```

This serializer will be able to encode and decode every data respecting your json schema specification.

Note: Take note that we don't use classic defaults for `JsonEncode` and `JsonDecode`. Using `JSON_UNESCAPED_SLASHES` only makes sense if you can have JSON References in your data (not specification). However, using `false` for `JsonDecode` (which means not using associative array but `\stdClass` instead) is mandatory.

As an example of why it's mandatory, a JSON Schema could contain the following valid specification:

```
{
    "type": "object",
    "properties": {
        "foo": {
            "type": ["array", "object"]
        }
    }
}
```

When using associative array, it would be tricky (but feasible) to deal with data inside the array or object (need to detect if all keys are numerical). The main problem comes when dealing with an empty array or object. In this case, there is no possibility to know if it was an array or object, and in some cases, decoding and recoding this value (with no modification) will change the data.

CHAPTER 6

Multi schemas generation

Jane JsonSchema allows to generate multiple schemas at the same time with different namespaces and directories to handle JSON References on others schemas.

6.1 Configuration

In order to use this feature, configuration of the `.jane` file will require a mapping of JSON Schema specification file linked to a root class, namespace and directory.

As an example you may have this:

```php
<?php

return [
    'mapping' => [
        __DIR__ . '/schema1.json' => [
            'root-class' => 'Foo',
            'namespace' => 'Vendor\Library\FooSchema',
            'directory' => __DIR__ . '/generated/Schema1',
        ],
        __DIR__ . '/schema2.json' => [
            'root-class' => 'Bar',
            'namespace' => 'Vendor\Library\BarSchema',
            'directory' => __DIR__ . '/generated/Schema2',
        ],
    ],
];
```

Using this configuration, Jane JsonSchema will generate all class of the `schema1.json` and `schema2.json` specification. Also, all references between both schemas will use the specific namespace.

As an example, given that you have the `Foo` object in `schema1.json`:

```
{
    "type": "object",
    "properties": {
        "foo": { "type": "string" }
    }
}
```

And the `Bar` one in `schema2.json`:

```
{
    "type": "object",
    "properties": {
        "bar": { "$ref": "schema1.json#" }
    }
}
```

The property `bar` of the `Bar` object will reference the `Vendor\Library\Schema1\Foo` class.

Note: If we don't specify the `schema1.json` in this configuration, Jane JsonSchema will still fetch the specification and generate the `Foo` class. However, it will be under the same namespace (`Vendor\Library\BarSchema`), and will have `BarBar` as the class name, instead of the `Foo` one.

6.2 Usage

In this case, Jane JsonSchema will generate two distinct `NormalizerFactory`, to be able to use references between schemas, you will only need to merge normalizers:

```
<?php

$normalizers = array_merge(
    \Vendor\Library\FooSchema\Normalizer\NormalizerFactory::create(),
    \Vendor\Library\BarSchema\Normalizer\NormalizerFactory::create()
);
```

CHAPTER 7

Json Schema

Jane OpenAPI is a library to generate, in PHP, an http client and its associated models and serializers from a OpenAPI specification: version 2.

7.1 Installation

Add this library with composer as a `dev` dependency:

```
composer require --dev jane-php/open-api "^4.0"
```

This library contains a lot of dependencies, to be able to generate code, which are not needed on runtime. However, the generated code depends on other libraries and a few classes that are available through the runtime package. It is highly recommended to add the runtime dependency as a requirement through composer:

```
composer require jane-php/open-api-runtime "^4.0"
```

By default, generated code is not formatted, to make it compliant to PSR2 standard and others format norms, you can add the PHP CS Fixer library to your dev dependencies (and it makes it easier to debug!):

```
composer require --dev friendsofphp/php-cs-fixer "^2.7.3"
```

CHAPTER 8

Generating a Client

This library provided a PHP console application to generate the Model, you can use it by executing the following command at the root of your project:

```
php vendor/bin/jane-openapi generate
```

This command will try to read a config file named `.jane-openapi` located on the current working directory. However, you can name it as you like and use the `--config-file` option to specify its location and name:

```
php vendor/bin/jane-openapi generate --config-file=jane-openapi-configuration.php
```

Note: No others options can be passed to the command. Having a config file ensure that a team working on the project always use the same set of parameters and, when it changes, give vision of the new option used to generate the code.

8.1 Configuration file

The configuration file consists of a simple PHP script returning an array:

```php
<?php

return [
    'openapi-file' => __DIR__ . '/openapi.json',
    'namespace' => 'Vendor\Library\Api',
    'directory' => __DIR__ . '/generated',
];
```

This example shows the minimum configuration required to generate a client:

- `openapi-file`: Specify the location of your OpenApi file, it can be a local file or a remote one `https://my.domain.com/my-api.json`. It can also be a `yaml` file.

- `namespace`: Root namespace of all of your generated code
- `directory`: Directory where the code will be generated

Given this configuration, you will need to add the following configuration to composer, in order to load the generated files:

```
"autoload": {
    "psr-4": {
        "Vendor\\Library\\Api\\": "generated/"
    }
}
```

8.2 Options

Other options are available to customize the generated code:

- `reference`: A boolean which indicate to add the support for JSON Reference into the generated code.
- `date-format`: A date format to specify how the generated code should encode and decode `\DateTime` object to string
- `strict`: A boolean which indicate strict mode (true by default), not strict mode generate more permissive client not respecting some standards (nullable field as an example)
- `async`: A boolean (false by default) which allows to generate a full asynchronous client using Amp with Artax client, see *Asynchronous Client* fore more information.
- `use-fixer`: A boolean which indicate if we make a first cs-fix after code generation
- `fixer-config-file`: A string to specify where to find the custom configuration for the cs-fixer after code generation
- `use-cacheable-supports-method`: A boolean which indicate if we use `CacheableSupportsMethodInterface` interface to improve caching performances when used with Symfony Serializer.

CHAPTER 9

Using a generated client

Generating a client will produce same classes as the *Json Schema* library:

- Model files in the `Model` namespace
- Normalizer files in the `Normalizer` namespace
- A `NormalizerFactory` class in the `Normalizer` namespace

Furthermore, it generates:

- Endpoints files in the `Endpoint` namespace, each API Endpoint will generate a class containing all the logic to go from Object to Request, and from Response to Object
- Client file in the root namespace containing all API endpoints
- ClientAsync file in the root namespace containing all API endpoints with an Async API. (only present if using the `'async' => true` option)

9.1 Creating the API Client

Generated `Client` class have a static method `create` which act like a factory to create your Client:

```php
<?php

$apiClient = Vendor\Library\Generated\Client::create();
```

Optionally, you can pass a custom `HttpClient` respecting the HTTPlug standard. If you which to use the constructor to reuse existing instances, sections below describe the 4 services used by it and how to create them.

9.2 Creating the Http Client

The main dependency on the `Client` class is an Http Client respecting the HTTPlug standard. We highly recommend you to read the docs on HTTPlug. This HTTP Client MAY redirect on a 3XX responses (depend on your API), but it

MUST not throw errors on 4XX and 5XX responses, as this can be handle by the generated code directly.

Recommended way of creating an HTTP Client is by using the discovery library of HTTPlug to create the client:

```
<?php

$httpClient = Http\Discovery\HttpClientDiscovery::find();
```

This allows user of the API to use any client respecting the standard.

9.3 Creating the Serializer

Like in *Using a generated Model*, creating a serializer is done by using the `NormalizerFactory` class:

```
<?php

$normalizers = Vendor\Library\Generated\Normalizer\NormalizerFactory::create();
$encoders = [new Symfony\Component\Serializer\Encoder\JsonEncoder(
    new Symfony\Component\Serializer\Encoder\JsonEncode(JSON_UNESCAPED_SLASHES),
    new Symfony\Component\Serializer\Encoder\JsonDecode(false))
];

$serializer = new Symfony\Component\Serializer\Serializer($normalizers, $encoders);
```

9.4 Creating the Message Factory

The generated endpoints will also need a service to transform parameters and object of the endpoint to a PSR7 Request This is done by using the Message Factory Interface from HTTPlug.

Like the HTTP Client, it is recommended to use the discovery library of HTTPlug to create it:

```
<?php

$messageFactory = Http\Discovery\MessageFactoryDiscovery::find();
```

9.5 Creating the Stream Factory

The generated endpoints will also need a service to transform body parameters like `resource` or `string` into *PSR7 Stream* when uploading file (multipart form). This is done by using the *Stream Factory Interface* from HTTPlug.

Like the HTTP Client and Message Factory, it is recommended to use the discovery library of HTTPlug to create it:

```
<?php

$streamFactory = Http\Discovery\StreamFactoryDiscovery::find();
```

9.6 Using the API Client

Generated code has complete PHPDoc comment on each method, which should correctly describe the endpoint. Method names for each endpoint depends on the `operationId` property of the OpenAPI specification. And if not present it will be generated from the endpoint path:

```php
<?php

$apiClient = Vendor\Library\Generated\Client::create();
// Operation id being listFoo
$foos = $apiClient->listFoo();
```

Also depending on the parameters of the endpoint, it may have 2 to more arguments.

Last parameter of each endpoint, allows to specify which type of data the method must return. By default, it will try to return an object depending on the status code of your response. But you can force the method to return a PSR7 Response object:

```php
$apiClient = Vendor\Library\Generated\Client::create();
// First argument is an empty list of parameters, second one being the return type
$response = $apiClient->listFoo([], Vendor\Library\Generated\Client::FETCH_RESPONSE);
```

This allow to do custom work when the API does not return standard JSON body.

9.7 Host and basePath support

Jane OpenAPI will never generate the complete url with the host and the base path for an endpoint. Instead, it will only do a request on the specified path.

If host and/or base path is present in the specification it is added, via the `PluginClient`, `AddHostPlugin` and `AddPathPlugin` thanks to HTTPlug plugin system when using the static `create`.

This allow you to configure different host and base path given a specific environment / server, which may defer when in test, preprod and production environment.

Jane OpenAPI will always try to use `https` if present in the scheme (or if there is no scheme). It will use the first scheme present if `https` is not present.

CHAPTER 10

Asynchronous Client

When setting the `async` option to true in the configuration, Jane OpenAPI will generate a full asynchronous API client. Using the Amp project with the Artax library.

A different client is necessary as it doesn't use the PSR7 standard which is not compatible when doing asynchronous execution in PHP.

Note: Long story short, this is mainly due to the `StreamInterface` of PSR7 standard not providing read and write methods compatible API, as they are blocking (the read method needs to return actual data, and cannot provide some sorts of callback / Promise / Coroutine data necessary when doing asynchronous operations)

10.1 Creating the Api Client

Like the `Client` class, the `ClientAsync` class have a static method `create` acting like a factory:

```php
<?php

$apiClientAsync = Vendor\Library\Generated\ClientAsync::create();
```

A custom `Amp\Artax\Client` can be passed to this method, also you can customize the `Serializer` by using the constructor:

```php
<?php

$normalizers = Vendor\Library\Generated\Normalizer\NormalizerFactory::create();
$encoders = [new Symfony\Component\Serializer\Encoder\JsonEncoder(
    new Symfony\Component\Serializer\Encoder\JsonEncode(JSON_UNESCAPED_SLASHES),
    new Symfony\Component\Serializer\Encoder\JsonDecode(false))
];

$serializer = new Symfony\Component\Serializer\Serializer($normalizers, $encoders);
```

(continues on next page)

(continued from previous page)

```
$httpClient = new Amp\Artax\DefaultClient();

$apiClientAsync = new Vendor\Library\Generated\ClientAsync($httpClient, $serializer);
```

10.2 Usage

If it's the first time using Amp, it is highly recommended to read the documentation about it. Using endpoints of your Api is exactly the same as the `Client` at the exception of returning an `Amp\Promise`:

```
<?php

Amp\Loop::run(function () {
    $apiClientAsync = Vendor\Library\Generated\ClientAsync::create();
    // Await the promise to get the final object
    $object = yield $apiClientAsync->getFoo(1);

    // Parallel api calls
    $objects = yield [
        $apiClientAsync->getFoo(1),
        $apiClientAsync->getFoo(2),
    ];
});
```

CHAPTER 11

Extending the Client

Some endpoints need sometimes custom implementation that were not possible to generate through the OpenAPI Specification. Jane OpenAPI try to be nice with this and each specific behavior of an API call has been seprated into different methods which are public or protected.

As an exemple you may want to encode in base64 a specific query parameter of an Endpoint. First step is to create your own Endpoint extending the generated one:

```php
<?php

namespace MyApi\Endpoint;

use MyApiGenerated\Endpoint\FooEndpoint as BaseEndpoint;
use Symfony\Component\OptionsResolver\Options;
use Symfony\Component\OptionsResolver\OptionsResolver;

class FooEndpoint extends BaseEndpoint
{
    protected function getQueryOptionsResolver(): OptionsResolver
    {
        $optionsResolver = parent::getQueryOptionsResolver();
        $optionsResolver->setNormalizer('bar', function (Options $options, $value) {
            return base64_encode($value);
        });

        return $optionsResolver;
    }
}
```

Once this endpoint is generated, you need to tell your Client to use yours endpoint instead of the Generated one. For that you can extends the generated client and override the method that use this endpoint:

```php
<?php

namespace MyApi;
```

(continues on next page)

(continued from previous page)

```
use MyApiGenerated\Client as BaseClient;
use MyApi\Endpoint\FooEndpoint;

class Client extends BaseClient
{
    public function getFoo(array $queryParameters = [], $fetch = self::FETCH_OBJECT)
    {
        return $this->executePsr7Endpoint(new FooEndpoint($queryParameters), $fetch);
    }
}
```

Then you will need to use your own client instead of the generated one. To extends other parts of the endpoint you can look at the generated code.