Jane Documentation Documentation

Release 6.0.0

Joel Wurtz

May 19, 2022

Contents

Jane is a set of libraries to generate Models & API Clients based on JsonSchema / OpenAPI specs by following high quality PHP code guidelines and respecting common & advanced PSR.

Jane also includes the AutoMapper, an advanced serializer that generates classes that can serialize or deserialize to improve performance. You can read more about it on the *Component: AutoMapper* page.

If you don't know what JSON Schema or OpenAPI are, you should consider reading *Choose the component you need* to help you sort what you need and how to use them.

If you already know which component you need, you can read one of the following getting started to use Jane:

- *Getting started: using JSON Schema*
- *Getting started: using OpenAPI*

If you are a more advanced user, you can read the detailed components pages:

- *Component: JSON Schema*
- *Component: OpenAPI*

Lastly, if you want to contribute there is some details about *How Jane works*, *Backwards compatibility* and *How our test suite works*

CHAPTER 1

Choose the component you need

Before using Jane you have to understand the two schema descriptors we are using:

- JSON Schema
- OpenAPI

Each of theses schema descriptors have their own use and you should choose the one that fit your use-case.

1.1 JSON Schema

"JSON Schema is a vocabulary that allows you to annotate and validate JSON documents". So it will allow you to describe your existing data format(s) and provides clear human- and machine- readable documentation.

Actually Jane does not support validation but it will come soon. With that descriptor you can describe documents and Jane will generate models & normalizers. This is mostly used when you need DTO or you need to use/store a JSON object that our normalizer can normalize/denormalize.

Hint: You can read more about what is JSON Schema and how to use it on the excellent "Understanding JSON Schema" book.

If you think this is what you need, you can read our getting started *Getting started: using JSON Schema*

Or you can see our detailled guides with more feature-focused stuff:

- *Elasticsearch models*
- *API Platform DTO*

1.2 OpenAPI

"The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection".

This descriptor is on top of JSON Schema, so every feature we have in JSON Schema will be in OpenAPI. And Jane will also generate a Client, endpoints and needed exceptions (for HTTP error responses). This is used with any API Client, some API even provide this file, like Stripe or Slack

Hint: You can read more about what is OpenAPI and how to use it on API Handyman tutorials.

If you think this is what you need, you can read our getting started *Getting started: using OpenAPI*

Or you can see our detailled guides with more feature-focused stuff:

- *External API Client*
- *Between two Symfony apps*

CHAPTER 2

Getting started: using JSON Schema

Jane JSON Schema is a library to generate models and serializers in PHP from a JSON Schema draft 2019-09.

2.1 Installation

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

```
composer require --dev jane-php/json-schema
```

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
```

With Symfony ecosystem, we created a recipe to make it easier to use Jane. You just have to allow contrib recipes before installing our packages:

```
composer config extra.symfony.allow-contrib true
```

Then when installing `jane-php/json-schema`, it will add all required files:

- `bin/json-schema-generate`: a binary file to run JSON Schema generation based on `config/jane/json-schema.php` configuration.
- `config/jane/json-schema.php`: your Jane configuration (see "Configuration file")
- `config/packages/json-schema.yaml`: Symfony Serializer configured to be optimized for Jane

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
```

2.2 Generating

This library provides 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: If you are using Symfony recipe, this command is embbeded in the `bin/json-schema-generate` binary file, you only have to run it to make it work

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(s) used to generate the code.

2.3 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 setup the PSR-4 autoload for the generated files:

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

For more details about generating JSON Schema, you can read "*Component: JSON Schema*" documentation.

2.4 Using

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 generate a `JaneObjectNormalizer` class that will act as an usual Symfony Normalizer that will lazy-load any needed normalizers.

Given this configuration:

```php
<?php

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

To use it out of Symfony ecosystem, you will have to do this:

```php
<?php

$normalizers = [
    new \Symfony\Component\Serializer\Normalizer\ArrayDenormalizer(),
    new \Vendor\Library\Generated\Normalizer\JaneObjectNormalizer(),
];

$serializer = new \Symfony\Component\Serializer\Serializer($normalizers, [new␣
↪\Symfony\Component\Serializer\Encoder\JsonEncoder()]);
$serializer->deserialize('{...}');
```

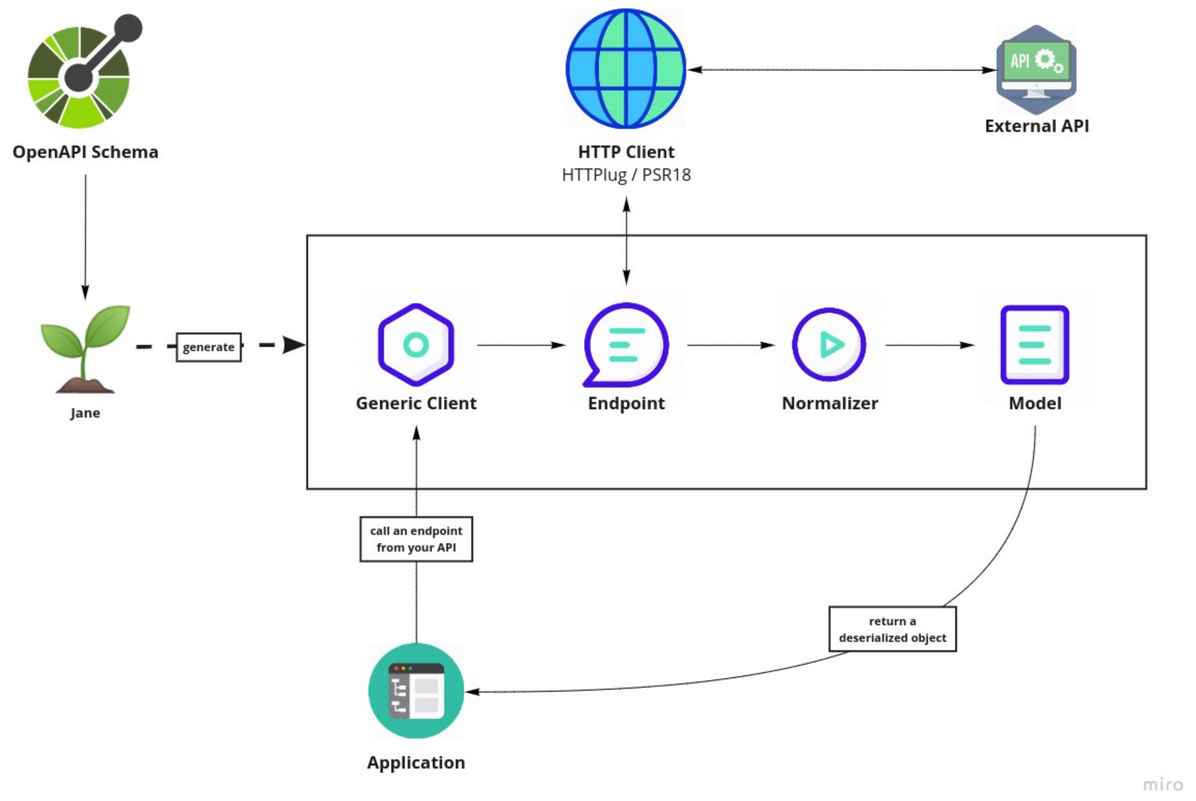
With Symfony ecosystem, you just have to use the recipe and all the configuration will be added automatically. This serializer will be able to encode and decode every data respecting your JSON Schema specification thanks to autowiring of the generated normalizers.

CHAPTER 3

Getting started: using OpenAPI

Jane OpenAPI is a library to generate, in PHP, an http client and its associated models and serializers from a OpenAPI specification: version 2 or 3.

Here is a quick schema to understand what Jane does and how does it work with your APIs



From left to right, Jane is gonna take your OpenAPI specification and generate files

- Generic client will be your starting point for your API, it will contains a `create` method to initialize everything we need and will have methods for all your API endpoints;
- Endpoint will be generated corresponding to all GET / POST / PUT / . . . endpoints your declared, they will be called in the Client instance methods;
- Normalizer will allow to convert from array to object and reverse, based on your models specification;
- Model are you model specification as PHP classes.

3.1 Installation

Jane supports both OpenAPI v2 & v3. Depending on your OpenAPI version, the command line will detect which version to use and if this version is actually installed in your dependencies.

You have to add the generation library as a `dev` dependency. 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. Choose your library depending on OpenAPI version you need (you can even install both if you want):

```
# OpenAPI 2
composer require --dev jane-php/open-api-2
composer require jane-php/open-api-runtime

# OpenAPI 3
composer require --dev jane-php/open-api-3
composer require jane-php/open-api-runtime
```

With Symfony ecosystem, we created a recipe to make it easier to use Jane. You just have to allow contrib recipes before installing our packages:

```
composer config extra.symfony.allow-contrib true
```

Then when installing `jane-php/open-api-*`, it will add all required files:

- `bin/open-api-generate`: a binary file to run JSON Schema generation based on `config/jane/open-api.php` configuration.
- `config/jane/open-api.php`: your Jane configuration (see “Configuration file”)
- `config/packages/open-api.yaml`: Symfony Serializer configured to be optimized for Jane

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
```

3.2 Generating

This library provides 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: If you are using Symfony recipe, this command is embbeded in the `bin/jane-open-api-generate` binary file, you only have to run it to make it work

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(s) used to generate the code.

3.3 Configuration file

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

```
<?php

return [
    'openapi-file' => __DIR__ . '/open-api.json',
    'namespace' => 'Vendor\Library\Generated',
    '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\\Generated\\": "generated/"
    }
}
```

For more details about generating JSON Schema, you can read "*Component: OpenAPI*" documentation.

3.4 Using

Generating a client will produce same classes as the *Getting started: using JSON Schema* library:

- Model files in the `Model` namespace
- Normalizer files in the `Normalizer` namespace

- A `JaneObjectNormalizer` 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 with the generated Normalizer
- `Client` file in the root namespace containing all API endpoints

3.5 Creating the API Client

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

```
<?php

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

Note: If you are using Symfony recipe, the client will be autowired. So you can use it anywhere by using your Client class

3.6 Creating the Serializer

Like in *Getting started: using JSON Schema*, creating a serializer is done by using the `JaneObjectNormalizer` class:

```
<?php

$normalizers = [
    new \Symfony\Component\Serializer\Normalizer\ArrayDenormalizer(),
    new \Vendor\Library\Generated\Normalizer\JaneObjectNormalizer(),
];

$serializer = new \Symfony\Component\Serializer\Serializer($normalizers, [new␣
↪\Symfony\Component\Serializer\Encoder\JsonEncoder()]);
$serializer->deserialize('{...}');
```

With Symfony ecosystem, you just have to use the recipe and all the configuration will be added automatically. This serializer will be able to encode and decode every data respecting your OpenAPI specification thanks to autowiring of the generated normalizers.

3.7 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

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

(continues on next page)

(continued from previous page)

```php
// Operation id being listFoo
$foos = $apiClient->listFoo();
```

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

For more details about using OpenAPI, you can read “*Component: OpenAPI*” documentation.

CHAPTER 4

Elasticsearch models

Here is a demo Symfony application of Jane with Elasticsearch integration.

You can find the fully working example on this repository: janephp/demo-with-elasticsearch.

4.1 Indexer

First, we need to index entities into Elasticsearch, to do that I made a command you can find in `project/src/Command/IndexCommand.php`. Here is the same code, decomposed to explain each steps:

```
// Here, $client is an instance of JoliCode\Elastically\Client, we use this library␣
↪on top of Elastica to create
// indexes, send documents, send requests and read results.
// With that `getIndexBuilder` method, we get a class to build indexes.
$indexBuilder = $client->getIndexBuilder();
// We create an index called "beers" (with date suffix)
$index = $indexBuilder->createIndex(self::BEERS_INDEX);
// We update the "beers" alias with this new index
$indexBuilder->markAsLive($index, self::BEERS_INDEX);

// Class to index our documents
$indexer = $client->getIndexer();

// We fetch all beers from database
$beers = $this->beerRepository->findAll();
foreach ($beers as $beer) {
    // For each entity, we convert it to a Generated\Model\Beer DTO
    $model = $this->autoMapper->map($beer, \Generated\Model\Beer::class);
    // We put it in a Document in order to index it
    $document = new \Elastica\Document($beer->getId(), $model);
    // And we schedule the Document to "beers" index
    $indexer->scheduleIndex(self::BEERS_INDEX, $document);
}
```

(continues on next page)

(continued from previous page)

```
// Flush all schedule documents & refresh "beers" index
$indexer->flush();
$indexer->refresh(self::BEERS_INDEX);
```

4.2 Controller

Then you can see in `project/src/Controller/BeerController.php` file some interaction to show Elasticsearch results. Same as before, decomposed code to explain each step:

```
// With the `getIndex` method, we get a reference of the index we want (here I'm␣
↪asking for 'beers' index)
$index = $client->getIndex(self::BEERS_INDEX);
// And we make a search query on the index (no arguments means we search for any␣
↪result)
$resultSet = $index->search();
// We get the results for given $resultSet
$results = $resultSet->getResults();

$output = ['beers' => []];
foreach ($results as $result) {
    // Then we get the model for each result
    // Here, thanks to Elastically and the Symfony serializer, the `getModel`
    // method will return a Generated\Model\Beer instance
    $output['beers'][] = $result->getModel();
}

return $this->json($output);
```

CHAPTER 5

API Platform DTO

Here is a demo Symfony application of Jane with API Platform integration.

You can find the fully working example on this repository: janephp/demo-with-apiplatform.

Disclaimer: This documentation is not a guide for API Platform, if you want more details about it, please consult their documentation

5.1 Resource

API Platform does support a way to have custom representation for our input or output. In this demo application, we focus on using a Jane model as our output model.

First, you will need to specify this model in your resource configuration:

```yaml
resources:
  App\Entity\Beer:
    attributes:
      output: Generated\Model\BeerOutput
```

In this configuration we specify the class used for our resource output. You can do more with this feature such as custom input class, read more on related documentation.

5.2 DataTransformer

Then, we need a DataTransformer to transform from the `App\Entity\Beer` entity to a `Generated\Model\BeerOutput` model. Here is this transformer, called `BeerOutputDataTransformer`, but decomposed to explain each steps:

```php
namespace App\DataTransformer;
```

(continues on next page)

(continued from previous page)

```php
use ApiPlatform\Core\DataTransformer\DataTransformerInterface;
use App\Entity\Beer;
use Generated\Model\BeerOutput;
use Jane\Component\AutoMapper\AutoMapperInterface;

class BeerOutputDataTransformer implements DataTransformerInterface
{
    private AutoMapperInterface $autoMapper;

    // Here we inject Jane AutoMapper, it's used to make the entity to model␣
↪transformation
    public function __construct(AutoMapperInterface $autoMapper)
    {
        $this->autoMapper = $autoMapper;
    }

    /**
     * @param Beer $data
     *
     * @return BeerOutput
     */
    public function transform($data, string $to, array $context = [])
    {
        // Will transformer our `App\Entity\Beer` entity to a␣
↪`Generated\Model\BeerOutput` model
        // thanks to the AutoMapper
        return $this->autoMapper->map($data, BeerOutput::class, $context);
    }

    /**
     * {@inheritdoc}
     */
    public function supportsTransformation($data, string $to, array $context = []):␣
↪bool
    {
        // Tells to use the `transform` method only if our data is a␣
↪`App\Entity\Beer` entity and if target model class
        // is `Generated\Model\BeerOutput`.
        return BeerOutput::class === $to && $data instanceof Beer;
    }
}
```

With only both of theses, you will have clean custom model output with API Platform!

CHAPTER 6

External API Client

Here is a demo Symfony application of Jane with an external API integration. We will see a working example of OpenApi v3 client onto a simple API that gives facts about cats and comment it. You can find this API documentation on following url: https://alexwohlbruck.github.io/cat-facts/.

You can find the fully working example on this repository: janephp/demo-external-api.

6.1 OpenAPI schema

First, we need a valid OpenAPI schema. Since this API doesn't have one I made my own, for some big API, there is existing OpenAPI schema, but be carefull with theses, they're often really big and you won't use all endpoints and models. A solution to this is to use the `whitelisted-paths` option in Jane configuration, or you can write your own schema to have only endpoints and models you need.

Here is the schema I made:

```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: 'CatFacts API'
servers:
  - url: https://cat-fact.herokuapp.com
paths:
  /facts/random:
    get:
      operationId: randomFact
      responses:
        200:
          description: 'Get a random `Fact`'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Fact'
```

(continues on next page)

(continued from previous page)

```
components:
  schemas:
    Fact:
      type: object
      properties:
        _id:
          type: string
          description: 'Unique ID for the `Fact`'
        __v:
          type: integer
          description: 'Version number of the `Fact`'
        user:
          type: string
          description: 'ID of the `User` who added the `Fact`'
        text:
          type: string
          description: 'The `Fact` itself'
        updatedAt:
          type: string
          format: date-time
          description: 'Date in which `Fact` was last modified'
        sendDate:
          type: string
          description: 'If the `Fact` is meant for one time use, this is the date␣
↪that it is used'
        deleted:
          type: boolean
          description: 'Weather or not the `Fact` has been deleted (Soft deletes are␣
↪used)'
        source:
          type: string
          description: 'Can be `user` or `api`, indicates who added the fact to the DB
↪'
        used:
          type: boolean
          description: 'Weather or not the `Fact` has been sent by the CatBot. This␣
↪value is reset each time every `Fact` is used'
        type:
          type: string
          description: 'Type of animal the `Fact` describes (e.g. ‘cat’, ‘dog’,␣
↪‘horse’)'
```

6.2 Declaring services

Here we will create services for Symfony. When requiring package `jane-php/open-api-3`, a recipe will be installed, it contains a `config/packages/jane.yaml` file, this file will contains wiring for the Normalizer, I added service for the API client here:

```
services:
    _defaults:
        autowire: true
        autoconfigure: true

    CatFacts\Api\Normalizer\JaneObjectNormalizer: ~
```

(continues on next page)

(continued from previous page)

```
    CatFacts\Api\Client:
        factory: ['CatFacts\Api\Client', 'create']
        lazy: true
```

6.3 Using your client

Finally, we create a controller that will fetch the data from the API and show a twig template to show the fact on `/fact` url.

```
use CatFacts\Api\Client;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class FactController extends AbstractController
{
    // Here we will inject the Jane Client, this will allow us to recover the cat␣
↪fact from the API!
    public function index(Client $client)
    {
        // We will render our home template with the cat fact from the API
        // Thanks to the OpenAPI scheme, Jane knows where is the server `https://cat-
↪fact.herokuapp.com` and the path
        // to use, so we only have to call related operation (defined by␣
↪`operationId` in OpenAPI)
        // Jane will call the endpoint and return a list of `CatFacts\Api\Model\Fact`␣
↪models
        return $this->render('fact.html.twig', [
            'fact' => $client->randomFact(),
        ]);
    }
}
```

CHAPTER 7

Between two Symfony apps

Here is a demo of Jane interacting with two Symfony apps. A frontend and an API apps.

You can find the fully working example on this repository: janephp/demo-between-two-apps.

7.1 A common contract

To make this all work we need a common contract, something that will declare our common model and how our API works. For this we will use OpenAPI 3, here is how this file will looks like:

```yaml
openapi: '3.0.2'
info:
  title: Between two apps
  description: Simple OpenAPI
  version: 1.0.0
servers:
  - url: 'http://api/'
paths:
  /beers:
    get:
      summary: Get beers
      operationId: getBeers
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Beer'
components:
  schemas:
```

(continues on next page)

(continued from previous page)

```
    Beer:
      type: object
      properties:
        name:
          type: string
        brewer:
          type: string
        style:
          type: string
        color:
          type: string
        alcohol:
          type: integer
```

Thanks to this schema, we know which endpoint will contains our data, where our server is (I'm using `http://api/` here because we are in docker environment and this is the service name) and how our data is structured.

7.2 API

First, in the API, we need an endpoint with a list of Beer models (like we described in our OpenAPI file). We will add a `BeerController` and make routing point path `/beers` to it. In this controller, we will list all beers and send them as JSON. Here is the controller code, decomposed to explain each steps:

```
namespace App\Controller;

use App\Entity\Beer as BeerEntity;
use App\Repository\BeerRepository;
use Jane\Component\AutoMapper\AutoMapperInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Generated\Model\Beer as BeerModel;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

class BeerController extends AbstractController
{
    public function list(BeerRepository $beerRepository, AutoMapperInterface
↪$autoMapper, NormalizerInterface $normalizer)
    {
        // Fetch all beers from database
        $beers = $beerRepository->findAll();

        // Will map all our beers from the entity `App\Entity\Beer` to the model␣
↪`Generated\Model\Beer`
        // For each entity, we use the AutoMapper to make this conversion
        $beerModels = \array_map(function (BeerEntity $beer) use ($autoMapper) {
            return $autoMapper->map($beer, BeerModel::class);
        }, $beers);

        // Return a response with `application/json` content-type from a list of␣
↪`Generated\Model\Beer` models
        // We use the normalizer to transform this list to an array of data
        return new JsonResponse($normalizer->normalize($beerModels));
    }
}
```

This is only our endpoint, we also have some configuration to generate Jane models (see `project/api/config/jane/`), entity, repository … You can see everything in `project/api/`.

7.3 Frontend

Then in our frontend part, we will recover data from the API and show them thanks a quick twig template. Here is the home controller code, even if he's really small, I would like to describe some stuff:

```
namespace App\Controller;

use Generated\Client;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class HomeController extends AbstractController
{
    // Here we will inject the Jane Client, this will allow us to recover beers from␣
↪the API!
    public function index(Client $client)
    {
        // We will render our home template with the beers from the API
        // Thanks to the OpenAPI scheme, Jane knows where is the server `http://api/`␣
↪and the path to use, so we only
        // have to call related operation (defined by `operationId` in OpenAPI)
        // Jane will call the endpoint and return a list of `Generated\Model\Beer`␣
↪models
        return $this->render('home.html.twig', [
            'beers' => $client->getBeers()
        ]);
    }
}
```

This will gives us all our data and render them, but we miss a thing! How this client was injected there? So here is the `project/front/config/packages/jane.yaml` file, that contains all Jane related configuration:

```
services:
  _defaults:
    autowire: true
    autoconfigure: true
    public: false

  # This is the usual Normalizer service, it's used to get all Jane generated␣
↪normalizers
  Generated\Normalizer\JaneObjectNormalizer: ~

  # And here we create a service for the Jane Client based on Client factory
  Generated\Client:
    factory: ['Generated\Client', 'create']
    lazy: true
```

I only described you our home controller and specific Jane configuration, we also have all usual Symfony configuration and code that you can see in `project/front/`.

CHAPTER 8

Validation

Since Jane 7.2.0, you can use the JSON Schema validation specification for your Jane models.

The goal of this feature is to allow your data to be validated based on your schema (either JSON Schema or OpenAPI) and in both ways (if you send data or if you receive data).

At the moment Jane supports most of the specification except for `minContains` and `maxContains` for the Array constraints. And for `dependentRequired` in the Object constraints.

8.1 Usage

For this example we will take the following JSON Schema as our base model:

```json
{
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "maxLength": 128
        },
        "alcohol": {
            "type": "number",
            "minimum": 0.0
        },
        "year": {
            "type": "integer",
            "minimum": 1000,
            "maximum": 2022
        }
    }
}
```

From what you can see here, we have some validation rules used in that schema for each properties. To enable the validation features in Jane, you'll have to set the `validation` parameter to `true` as following:

```php
<?php

return [
    'json-schema-file' => __DIR__ . '/beer.json',
    'root-class' => 'Beer',
    'namespace' => 'App\Generated',
    'directory' => __DIR__ . '/generated',
    'validation' => true,
];
```

With that configuration, validators will be generated and during normalization or denormalization they will be called to validate your data. You have nothing to change of your usual Jane calls since all the logic is put into the generated Normalizer.

So with the following code:

```php
<?php

$normalizer = new BeerNormalizer();
$model = $normalizer->denormalize([
    'name' => 'Kasteel Tripel',
    'alcohol' => 11.0,
    'year' => 1811,
], Beer::class);

dump($model);
```

You will get that model which was validated before returning:

```
^ App\Generated\Model\Beer^ {#107273
  #name: "Kasteel Tripel"
  #alcohol: 11.0
  #year: 1811
}
```

In my example, I was using only valid values for all my properties, but if I change the year field to the `811` value, it will cause an error. No model will be returned and an exception will be thrown as following:

```
^ Jane\Component\JsonSchema\Tests\Validation\Generated\Validator\ValidationException^
↪{#112477
  #message: "Model validation failed with 1 errors."
  #code: 400
  #file: "./src/Component/JsonSchema/Tests/Validation/Generated/Validator/
↪BeerValidator.php"
  #line: 13
  -violationList: Symfony\Component\Validator\ConstraintViolationList^ {#105008
    -violations: array:1 [
      0 => Symfony\Component\Validator\ConstraintViolation^ {#114455
        -message: "This value should be greater than or equal to 1000."
        -messageTemplate: "This value should be greater than or equal to {{ compared_
↪value }}."
        -parameters: array:3 [
          "{{ value }}" => "811"
          "{{ compared_value }}" => "1000"
          "{{ compared_value_type }}" => "float"
        ]
        -plural: null
```

(continues on next page)

(continued from previous page)

```
        -root: array:3 [
          "name" => "Kasteel Tripel"
          "alcohol" => 11.0
          "year" => 811
        ]
        -propertyPath: "[year]"
        -invalidValue: 811
        -constraint: Symfony\Component\Validator\Constraints\GreaterThanOrEqual^ {
↪#112504
          +payload: null
          +groups: array:1 [
            0 => "Default"
          ]
          +message: "This value should be greater than or equal to {{ compared_value }
↪}."
          +value: 1000.0
          +propertyPath: null
        }
        -code: "ea4e51d1-3342-48bd-87f1-9e672cd90cad"
        -cause: null
      }
    ]
  }
}
```

To generate validators we use the Symfony Validator component to have independent validator for each generated models. All the constraints for each models will be compiled in that validator and embedded in a Collection constraint that is used to describe all fields and constraints within that model.

For our Beer model, the validator will look like this:

```php
<?php

namespace App\Generated\Validator;

use Symfony\Component\Validator\Constraints;
use Symfony\Component\Validator\Validation;

class BeerValidator implements App\Generated\Validator\ValidatorInterface
{
    public function validate($data) : void
    {
        $constraints = array(
            new Constraints\Collection(array('fields' => array(
                'name' => new Constraints\Optional(array(
                    new Constraints\Length(array(
                        'max' => 128,
                        'maxMessage' => 'This value is too long. It should have {{␣
↪limit }} characters or less.'
                    )),
                    new Constraints\Type(array('string'))
                )),
                'alcohol' => new Constraints\Optional(array(
                    new Constraints\GreaterThanOrEqual(array('value' => 0.0)),
                    new Constraints\Type(array('float'))
                )),
                'year' => new Constraints\Optional(array(
```

(continues on next page)

(continued from previous page)

```
                    new Constraints\LessThanOrEqual(array('value' => 2022.0)),
                    new Constraints\GreaterThanOrEqual(array('value' => 1000.0)),
                    new Constraints\Type(array('integer')))
                )),
                'allowExtraFields' => true
            ))
        );

        $validator = Validation::createValidator();
        $violations = $validator->validate($data, $constraints);

        if ($violations->count() > 0) {
            throw new ValidationException($violations);
        }
    }
}
```

Here we can see that each field has a type constraint to check the value passed and if we have more constraints they will be added accordingly.

Inside the normalizer, you can find that validator usage with:

```
public function denormalize($data, $class, $format = null, array $context = array())
{
    $validator = new BeerValidator();
    $validator->validate($data);

    // ...
}
```

This allows the validation to be done without having anything else than the configuration to do. Also you can use the Validator out of Jane normalization by doing exactly the same as in the Normalizer.

CHAPTER 9

How to manage nullable properties

Most of the time, the schema you get from vendor have issues about nullability of their properties.

Here Jane has an option called `strict` mode, by default we will follow strictly your schema types. But you can switch to non-strict mode (by having the option set to false in your Jane configuration file: `"strict" => false`). By doing this, any property of your schema will be considered nullable.

You can also try to fix your schema, here are some tips depending on which schema you are using

9.1 JSON Schema

Jane is actually supporting JSON Schema draft 2019-09. Which support `null` type, you just have to add it to your type field as follows:

```
type:
  - 'null'    # Note the quotes around 'null'
  - string
```

9.2 OpenAPI v2 / Swagger

OpenAPI v2 does not support `null` type. Instead, most libraries does support `x-nullable` field in order to fake `null` support. You can use it as follows:

```
type: string
x-nullable: true
```

If you are using OpenAPI v2, consider migrating to OpenAPI v3 to get proper nullability support.

9.3 OpenAPI v3.0.x

OpenAPI v3 still does not support `null` type but added a `nullable` field in order to say that property is nullable or not. By default this field is set to `false`. You can use it as follows:

```yaml
type: string
nullable: true
```

9.4 OpenAPI v3.1.x

This new OpenAPI version will be compatible with JSON Schema. So everything will be the same as JSON Schema. You still can use `nullable` field but it will be deprecated in favor of JSON Schema `null` type.

As of 25/04/2020, this version is not yet released.

CHAPTER 10

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.

10.1 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.

10.2 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.

10.3 Runtime Libraries

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

CHAPTER 11

How Jane works

This documentation describes how JSON Schema & OpenAPI Jane libraries work to generate the code. It is mainly oriented for people wanting to contribute to theses libraries.

Theses libraries is based on 3 different steps:

11.1 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.

11.2 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.

11.3 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 12

How our test suite works

This documentation describes how our test suite works and how you can interact with it. It is mainly oriented for people wanting to contribute to theses libraries.

We test each components with a global composer configuration. On repository root you can find a `composer.json` file that contains all components dependencies, in our CI we install theses dependencies then we tests all components.

If you want to run it locally, you can do:

```
composer update
vendor/bin/phpunit
```

You can also run same commands to test a single component, you just have to cd inside the component first:

```
cd src/JsonSchema
composer update
vendor/bin/phpunit
```

We mainly use JsonSchema / OpenAPI fixtures for our tests. When we add a feature, we create a new folder in related component tests folder with a schema related to the added feature. That way when we run tests, it will generate a `generated/` folder that will be compared with a `expected/` that contains generated files like they should be.

If you just created a fixture folder and don't have `expected/` folder, just run tests and check manually `generated/` files and if everything is ok, you can copy the folder and name it `expected/`. If you have to do this on multiple fixtures, you can use `./replace-all-expected-fixtures.sh` script. It will copy all `generated/` into `expected/` folder. So please be sure that everything is okay before running this script.

By default, we don't run generated client related tests locally, because you need to run stoplightio/prism with configuration as following:

- `nohup prism mock -p 4010 -m src/OpenApi3/Tests/client/openapi.yaml &`
- `nohup prism mock -p 4011 -m src/OpenApi2/Tests/client/swagger.yaml &`

Both theses will run a "fake" API based on the given OpenApi scheme. If you want to see logs, you can remove `nohup` and `&` keywords on given commands. With theses servers running you can now run generated client tests with `vendor/bin/phpunit --exclude-group none` command. Even if we don't run theses tests locally, they will always run on CI.

CHAPTER 13

Component: JSON Schema

13.1 Introduction

Jane JsonSchema is a library to generate models and serializers in PHP from a JSON Schema draft 2019-09.

13.2 Installation

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

```
composer require --dev jane-php/json-schema
```

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 mandatory to add the runtime dependency as a requirement through composer:

```
composer require jane-php/json-schema-runtime
```

With Symfony ecosystem, we created a recipe to make it easier to use Jane. You just have to allow contrib recipes before installing our packages:

```
composer config extra.symfony.allow-contrib true
```

Then when installing `jane-php/json-schema`, it will add all the required files:

- `bin/json-schema-generate`: a binary file to run JSON Schema generation based on `config/jane/json-schema.php` configuration;
- `config/jane/json-schema.php`: your Jane configuration (see "Configuration file");
- `config/packages/json-schema.yaml`: Symfony Serializer configured to be optimized for Jane.

By default, generated code is not formatted, to make it compliant to PSR2 standard and others coding style formats, 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
```

13.3 Generating a Model

This library provides 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: If you are using Symfony recipe, this command is embbeded in the `bin/json-schema-generate` binary file, you only have to run it to make it work

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(s) used to generate the code.

Hint: If you have a really big specification and want to optimize your generation time, you can disable garbage collector during generation, you can read more about it on Scrutinizer blog post . To do that, use Jane as following: `php -d zend.enable_gc=0 vendor/bin/jane generate`.

13.3.1 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 setup the PSR-4 autoload for the generated files:

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

13.3.2 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-time format to specify how the generated code should encode and decode `\DateTime` object to string. This option is only for format `date-time`.
- `full-date-format`: A date format to specify how the generated code should encode and decode `\DateTime` object to string. This option is only for format `date`.
- `date-prefer-interface`: The `\DateTimeInterface` is the base of every `\DateTime` related action. This makes it more compatible with other DateTime libraries like Carbon. This option replace `\DateTime` returns with `\DateTimeInterface`, it's disabled by default.
- `date-input-format`: During denormalization (from array to object), we may have a different format than the output format. This option allows you to specify which format you want. By default it will take `date-format` configuration.
- `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) client.
- `use-fixer`: A boolean which indicate if we make a first cs-fix after code generation, is disabled by default.
- `fixer-config-file`: A string to specify where to find the custom configuration for the cs-fixer after code generation, will remove all Jane default cs-fixer default configuration.
- `clean-generated`: A boolean which indicate if we clean generated output before generating new files, is enabled by default.
- `use-cacheable-supports-method`: A boolean which indicate if we use `CacheableSupportsMethodInterface` interface to improve caching performances when used with Symfony Serializer.
- `skip-null-values`: When having nullable properties, you can enforce normalization to skip theses properties even if they are nullable. This option allows you to not have theses properties when they're not set (`null`). By default it is enabled.
- `skip-required-fields`: If your model has required fields, this option allows you to skip the required behavior that forces them to be present during denormalization. By default it is disabled.
- `validation`: Will enable validation following JSON Schema validation specification. By default it is disabled. You can read more about it on the dedicated guide: *Validation*.

13.4 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 generate a `JaneObjectNormalizer` class that will act as an usual Symfony Normalizer.

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 = [
    new \Symfony\Component\Serializer\Normalizer\ArrayDenormalizer(),
    new \Vendor\Library\Generated\Normalizer\JaneObjectNormalizer(),
];

$serializer = new \Symfony\Component\Serializer\Serializer($normalizers, [new␣
↪\Symfony\Component\Serializer\Encoder\JsonEncoder()]);
$serializer->deserialize('{...}');
```

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

13.5 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.

13.5.1 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__ . '/schema-foo.json' => [
            'root-class' => 'Foo',
            'namespace' => 'Vendor\Library\Foo',
            'directory' => __DIR__ . '/generated/Foo',
        ],
        __DIR__ . '/schema-bar.json' => [
            'root-class' => 'Bar',
```

(continues on next page)

(continued from previous page)

```
            'namespace' => 'Vendor\Library\Bar',
            'directory' => __DIR__ . '/generated/Bar',
        ],
    ],
];
```

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

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

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

And the `Bar` one in `schema-bar.json`:

```
{
    "type": "object",
    "properties": {
        "foo": { "$ref": "schema-foo.json#" }
    }
}
```

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

Note: If we don't specify the `schema-foo.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 `FooBar` as the class name, instead of the `Foo` one.

Note: If provided, the options `fixer-config-file`, `use-fixer` and `clean-generated` have to bee defined at the root level of the array and not in each mapping schema configuration.

13.5.2 Usage

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

```
$normalizers = [
    new \Symfony\Component\Serializer\Normalizer\ArrayDenormalizer(),
    new \Vendor\Library\Foo\Normalizer\JaneObjectNormalizer(),
    new \Vendor\Library\Bar\Normalizer\JaneObjectNormalizer(),
];

$serializer = new \Symfony\Component\Serializer\Serializer($normalizers, [new␣
↪\Symfony\Component\Serializer\Encoder\JsonEncoder()]);
$serializer->deserialize('{...}');
```

Note: With Symfony ecosystem, you just have to use the recipe and all the configuration will be added automatically. Both serializer will be able to encode and decode every data respecting your JSON Schema specification thanks to autowiring of the generated normalizers.

CHAPTER 14

Component: OpenAPI

Jane OpenAPI is a library to generate, in PHP, an HTTP client and its associated models and serializers from a OpenAPI specification: version 2 or 3. Jane supports both OpenAPI v2 & v3. Depending on your OpenAPI version, the command line will detect which version to use and if this version is actually installed in your dependencies.

14.1 Installation

Jane supports both OpenAPI v2 & v3. Depending on your OpenAPI version, the command line will detect which version to use and if this version is actually installed in your dependencies.

You have to add the generation library as a `dev` dependency. 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 mandatory to add the runtime dependency as a requirement. Choose your library depending on OpenAPI version you need (you can even install both if you want):

```
# OpenAPI 2
composer require --dev jane-php/open-api-2
composer require jane-php/open-api-runtime

# OpenAPI 3
composer require --dev jane-php/open-api-3
composer require jane-php/open-api-runtime
```

With Symfony ecosystem, we created a recipe to make it easier to use Jane. You just have to allow contrib recipes before installing our packages:

```
composer config extra.symfony.allow-contrib true
```

Then when installing `jane-php/open-api-*`, it will add all the required files:

- `bin/open-api-generate`: a binary file to run JSON Schema generation based on `config/jane/open-api.php` configuration;
- `config/jane/open-api.php`: your Jane configuration (see "Configuration file");

- `config/packages/open-api.yaml`: Symfony Serializer configured to be optimized for Jane.

By default, generated code is not formatted, to make it compliant to PSR2 standard and others coding style formats, 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
```

14.2 Generating a Client

This library provides 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: If you are using Symfony recipe, this command is embbeded in the `bin/open-api-generate` binary file, you only have to run it to make it work

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(s) used to generate the code.

Hint: If you have a really big specification and want to optimize your generation time, you can disable garbage collector during generation, you can read more about it on Scrutinizer blog post . To do that, use Jane as following: `php -d zend.enable_gc=0 vendor/bin/jane-openapi generate`.

14.2.1 Configuration file

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

```
<?php

return [
    'openapi-file' => __DIR__ . '/open-api.json',
    'namespace' => 'Vendor\Library\Generated',
    '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\\Generated\\": "generated/"
    }
}
```

14.2.2 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
- `date-format`: A date-time format to specify how the generated code should encode and decode `\DateTime` object to string. This option is only for format `date-time`.
- `full-date-format`: A date format to specify how the generated code should encode and decode `\DateTime` object to string. This option is only for format `date`.
- `date-prefer-interface`: The `\DateTimeInterface` is the base of every `\DateTime` related action. This makes it more compatible with other DateTime libraries like Carbon. This option replace `\DateTime` returns with `\DateTimeInterface`, it's disabled by default.
- `date-input-format`: During denormalization (from array to object), we may have a different format than the output format. This option allows you to specify which format you want. By default it will take `date-format` configuration.
- `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) client.
- `use-fixer`: A boolean which indicate if we make a first cs-fix after code generation, is disabled by default.
- `fixer-config-file`: A string to specify where to find the custom configuration for the cs-fixer after code generation, will remove all Jane default cs-fixer default configuration.
- `clean-generated`: A boolean which indicate if we clean generated output before generating new files, is enabled by default.
- `use-cacheable-supports-method`: A boolean which indicate if we use `CacheableSupportsMethodInterface` interface to improve caching performances when used with Symfony Serializer.
- `skip-null-values`: When having nullable properties, you can enforce normalization to skip theses properties even if they are nullable. This option allows you to not have theses properties when they're not set (`null`). By default it is enabled.
- `skip-required-fields`: If your model has required fields, this option allows you to skip the required behavior that forces them to be present during denormalization. By default it is disabled
- `validation`: Will enable validation following JSON Schema validation specification. By default it is disabled. You can read more about it on the dedicated guide: *Validation*.
- `whitelisted-paths`: This option allows you to generate only needed endpoints and related models. Be carefull, that option will filter models used by whitelisted endpoints and generate model & normalizer only for them. Here is some examples about how to use it:

```php
<?php

return [
    // your usual configuration ...
    'whitelisted-paths' => [
        '\/foo$',
        ['\/foo\/(bar|baz)'],
        ['\/foo$', 'GET'],
        ['\/foo$', ['POST']],
        ['\/foo$', ['POST', 'PUT']]
    ],
];
```

There is many ways to use it, first you atleast need a regex defining which endpoint is whitelisted. This endpoint can be either a string or in an array. If you don't provide any HTTP method, we will just accept any methods, but you can provide either a string or array as second argument to specify which method you accept.

- `endpoint-generator`: Generator Class which can specify custom endpoint interface & corresponding trait (this class should extends `\Jane\Component\OpenApi3\Generator\EndpointGenerator`)
- `custom-query-resolver`: This option allows you to customize the query parameter normalizer for each of the API endpoint with a userland callback. Here is all possible combinations:

```php
<?php

use App\BoolCustomQueryResolver;
use App\IntCustomQueryResolver;
use App\BarCustomQueryResolver;
use App\BazCustomQueryResolver;

return [
    // your usual configuration ...
    'custom-query-resolver' => [
        '__type' => [
            'bool' => BoolCustomQueryResolver::class,
            'int' => IntCustomQueryResolver::class,
        ],
        '/foo' => [
            'get' => [
                'bar' => BarCustomQueryResolver::class,
                'baz' => BazCustomQueryResolver::class,
            ],
            'post' => [
                'bar' => BarCustomQueryResolver::class,
            ],
        ],
    ],
];
```

There are many ways to use it. You can either use the `__type` key to specify a custom query normalizer for a given type (`bool`, `int`, `string`, `...`) and give it your class that contains the custom normalizer by extending the generated runtime `CustomQueryResolver` class. You can also filter the usage of your custom normalizer by giving the exact path, method and parameter name where you want to apply it.

- `throw-unexpected-status-code`: Will return a `UnexpectedStatusCodeException` if nothing has been matched during the transformation of the Endpoint body (including described exceptions). By default, it's disabled.
- `custom-string-format-mapping`: This option allows you to specify in which class a string property

will be deserialized according to it's format option. It can be used to customize a date-time field, or to add non supported formats. More details in the dedicated section.

14.3 Using a generated client

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

- Model files in the `Model` namespace
- Normalizer files in the `Normalizer` namespace
- A `JaneObjectNormalizer` 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 with the generated Normalizer
- `Client` file in the root namespace containing all API endpoints

14.4 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();
```

Note: If you are using Symfony recipe, the client will be autowired. So you can use it anywhere by using your Client class

Note: Optionally, you can pass a custom `HttpClient` respecting the PSR18 Client 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.

14.4.1 Creating the Http Client

The main dependency on the `Client` class is an HTTP client respecting the PSR18 client standard. We highly recommend you to read the PSR18 specification. 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 to create the client:

```php
<?php

$httpClient = Http\Discovery\Psr18ClientDiscovery::find();
```

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

Hint: You can use clients such as Symfony HttpClient as PSR18 client.

14.4.2 Creating the Request Factory

The generated endpoints will also need a factory to transform parameters and object of the endpoint to a PSR7 Request.

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

```
<?php

$requestFactory = Http\Discovery\Psr17FactoryDiscovery::findRequestFactory();
```

14.4.3 Creating the Serializer

Like in *Component: JSON Schema*, creating a serializer is done by using the `JaneObjectNormalizer` class:

```
<?php

$normalizers = [
    new \Symfony\Component\Serializer\Normalizer\ArrayDenormalizer(),
    new \Vendor\Library\Generated\Normalizer\JaneObjectNormalizer(),
];

$serializer = new \Symfony\Component\Serializer\Serializer($normalizers, [new␣
↪\Symfony\Component\Serializer\Encoder\JsonEncoder()]);
$serializer->deserialize('{...}');
```

With Symfony ecosystem, you just have to use the recipe and all the configuration will be added automatically. This serializer will be able to encode and decode every data respecting your OpenAPI specification thanks to autowiring of the generated normalizers.

14.4.4 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).

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

```
<?php

$streamFactory = Http\Discovery\Psr17FactoryDiscovery::findStreamFactory();
```

14.5 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.

14.5.1 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 php-http 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.

14.5.2 Having custom plugins

If you want to support more behavior such as authentication or other stuff that need a plugin, you can pass them through the second argument of the static `create` method.

14.5.3 Authentication

We do generate a plugin for each authentication method declared in your scheme. It does support:

- `apiKey` in header & query for both OpenAPI v2 & v3
- HTTP Basic & Bearer for OpenAPI v3

Quick example of how your authentication definition could look (OpenAPI v3):

```yaml
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    BearerAuth:
      type: http
      scheme: bearer
```

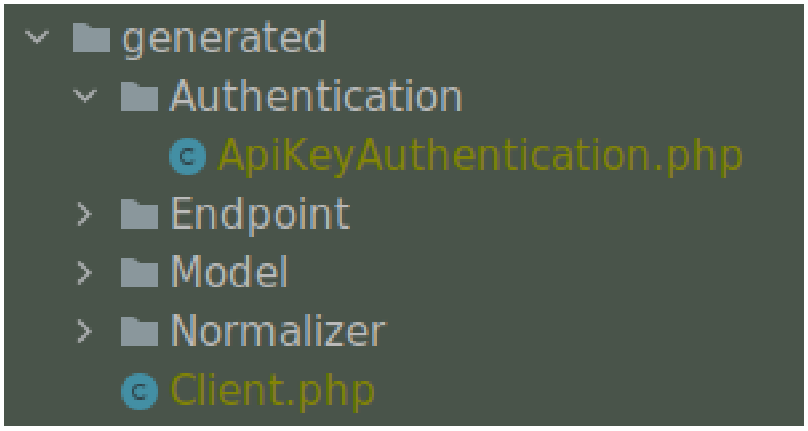
(continues on next page)

(continued from previous page)

```
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-Key
```

When your OpenAPI definition contains it, Jane will generate a Authentication namespace that contains all plugins you need for your API. Then you give all your authentication plugins to `Jane\Component\OpenApiRuntime\Client\Plugin\AuthenticationRegistry`. And finally you can pass it to your Jane Client (only if you let Jane make a HTTP Client for you, otherwise this second parameters is ignored).

An example Authentification directory:



```
generated
  Authentication
    ApiKeyAuthentication.php
  Endpoint
  Model
  Normalizer
  Client.php
```

This `AuthenticationRegistry` class is used to match security scopes in your API, if an Endpoint require a certain authentication method, then it will use it. You need to have `security` fields correctly made in your scheme in order to use this class. If they're not set, you can simply pass the authentication plugin to your Jane Client.

Here is how you can use it:

```
$authenticationRegistry = new AuthenticationRegistry([new ApiKeyAuthentication($this->
↪apiKey)]);
$client = Client::create(null, [$authenticationRegistry]);
$foo = $client->foo();
```

You can replace `Client::create` first argument with your custom HttpClient if needed as usual.

14.6 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

namespace Vendor\Library\Generated\Endpoint;

use Vendor\Library\Generated\Endpoint\FooEndpoint as BaseEndpoint;
```

(continues on next page)

(continued from previous page)

```php
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 Vendor\Library\Generated;

use Vendor\Library\Generated\Client as BaseClient;
use Vendor\Library\Generated\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.

14.7 Custom string formats

Jane support some strings format, but it can't support all of them because it's an open keyword. You may want to serialize a property to an UUID, or have a specific datetime format for a field (a datetime format that is not the same as the one configured with `date-format` or `full-date-format`.

To do so, you need to provide:

- while generating the client: an associative array for the key `custom-string-format-mapping`
- at runtime: one or more Normalizer (which implement `Symfony\Component\Serializer\Normalizer\Normalizer`

14.7.1 Example

Configuration file:

```php
<?php

return [
    'json-schema-file' => __DIR__ . '/json-schema.json',
    'root-class' => 'MyModel',
    'namespace' => 'Vendor\Library\Generated',
    'directory' => __DIR__ . '/generated',
    'custom-string-format-mapping' => [
        'uuid' => \Symfony\Component\Uid\UuidV4::class
    ]
];
```

Your OpenAPI schema:

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Example
paths:
  /some-path:
    get:
      summary: Get something
      operationId: getSomething
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Something"
components:
  schemas:
    Something:
      type: object
      required:
        - id
        - uuid
      properties:
        id:
          type: 'integer'
        uuid:
          type: 'string'
          # the following keyword is important
          format: 'uuid'
```

Usage of the generated client:

```php
<?php

$client = \Vendor\Library\Generated\Client::create(
    $httpClient,
    [], // additional http client plugins
    // additional normalizers
    [
        new \Symfony\Component\Serializer\Normalizer\UidNormalizer()
    ]
);
```

CHAPTER 15

Component: AutoMapper

15.1 Description

Jane AutoMapper is an experimental library that generate AutoMapper class which allows to automap values from Class to Class.

Taken from AutoMapper/AutoMapper:

AutoMapper is a simple little library built to solve a deceptively complex problem - getting rid of code that mapped one object to another. This type of code is rather dreary and boring to write, so why not invent a tool to do it for us?

In PHP libraries and application mapping from one object to another is fairly common:

- `ObjectNormalizer` / `GetSetMethodNormalizer` in `symfony/serializer`
- Mapping request data to object in `symfony/form`
- Hydrate object from SQL results in Doctrine
- Migrating legacy data to new model
- Mapping from database model to DTO objects (API / CQRS / . . .)
- And even more. . .

The goal of this component is to offer an abstraction on top of this subject. For that goal it provides an unique interface (other code is only implementation detail):

```
interface AutoMapperInterface
{
    /**
     * Map data from to target.
     *
     * @param array|object        $source   Any data object, which may be an object or␣
↪an array
     * @param string|array|object $target   To which type of data, or data, the source␣
↪should be mapped
```

(continues on next page)

(continued from previous page)

```
     * @param Context              $context Options mappers have access to
     *
     * @return array|object The mapped object
     */
    public function map($source, $target, Context $context = null);
}
```

The source is from where the data comes from, it can be either an array or an object. The target is where the data should be mapped to, it can be either a string (representing a type: array or class name) or directly an array or object (in that case construction of the object is avoided).

Current implementation handle all of those possiblities at the exception of the mapping from a dynamic object (array / stdClass) to another dynamic object.

15.2 Using AutoMapper

15.2.1 Basic usage

Someone who wants to map an object will only have to do this:

```
// With class name
$target = $automapper->map($source, Foo::class);
// With existing object
$target = new Foo();
$target = $automapper->map($source, $target);
// To an array
$target = $automapper->map($source, 'array');
// From an array
$source = ['a' => 'b'];
$target = $automapper->map($source, Foo::class);
```

15.2.2 With custom context

Context object allow to pass options for the mapping:

```
// Using context
$context = new Context();
$target = $automapper->map($source, Foo::class, $context);

// Groups (serializer annotation), will only map value that match those group in␣
↪source and target
$context = new Context(['groupA', 'groupB']);
// Allowed attributes, will only map specific properties (exclude others), allow␣
↪nesting for sub mapping like the serializer component
$context = new Context(null, ['propertyA', 'propertyB', 'foo' => ['fooPropertyA']]);
// Ignored attributes, exclude thos propreties include others
$context = new Context(null, null, ['propertyA', 'propertyB', 'foo' => ['fooPropertyA
↪']]);
// Set circular reference limit
$context->setCircularReferenceLimit(2);
// Set circular reference handler
$context->setCircularReferenceHandler(function () { ... });
```

15.3 Features

15.3.1 Nested Mapping

This component map nested class when it's possible.

15.3.2 Circular Reference

Default circular reference implementation is to keep them during mapping, which means somethings like:

```
$foo = new Foo();
$foo->setFoo($foo);

$target = $this->automapper->map($foo, 'array');
```

Will produce an array where the foo property will be a reference to the parent.

Having that allow using this component as a DeepCloning service by mapping to the same object:

```
$foo = new Foo();
$foo->setFoo($foo);

$deepClonedFoo = $this->automapper->map($foo, Foo::class);
```

15.3.3 Max Depth

This component understand the Max Depth Annotation of the Serializer component and will not map after it's reached.

15.3.4 Name Converter

Default implementation allows you to pass a Name Converter when converting to or from an array to change the property name used.

15.3.5 Discriminator Mapping

This component understand the Discriminator Mapping Annotation of the Serializer component and should correctly handle construction of object when having inheritance.

15.3.6 Type casting

This component will try to correctly map scalar values (going from int to string, etc).

15.3.7 Transformer extension

Sometimes you have to convert special objects (such as `\Money\Money` from `moneyphp\money` package), to do that you should create a custom TransformerFactory and its Transformers. We made an example in the AutoMapper tests files that you can look at.

To use a custom TransformerFactory class, you have to do as following:

```
$autoMapper->bindTransformer(new TransformerFactory());
```

With the Symfony bundle, you have to tag your TransformerFactory class with a `jane_auto_mapper.transformer_factory` tag. This will use automatically the TransformerFactory.

15.3.8 Skip null values

This context option allows us to ignore `null` values from source attributes. So if we use that option and our target object has a value, it will keep it.

Here is a quick example:

```
class Input
{
  public ?string $name = null;
}

class MyEntity
{
  private string $name;
  public function setName(string $name) {
    $this->name = $name;
  }
  public function getName() {
    return $this->name;
  }
}

$myEntity = new MyEntity();
$myEntity->setName('foobar');
$input = new Input();

$autoMapper->map($input, $myEntity, ['skip_null_values' => true]);
echo $myEntity->getName(); // "foobar"
```

15.4 Implementation

Default implementation use code generation for mapping, it reads once the metadata needed to build the mapper then write PHP code, after this, no metadata reading or analysis is done, only the generated mapper is used.

This allow for very fast mapping, here is some benchmarks using the library:

- travis-ci.org/idr0id/php-mappers-benchmarks/builds/361253808?utm_source=github_status&utm_medium=notification
- travis-ci.org/php-serializers/ivory-serializer-benchmark

And here is an example of generated code

15.5 Symfony Bundle

If you want to use the AutoMapper with Symfony, you can require the related bundle:

```
composer require jane-php/automapper-bundle
```

Then you have to add the bundle class in your `config/bundles.php` file:

```php
return [
    // ...
    Jane\Bundle\AutoMapperBundle\JaneAutoMapperBundle::class => ['all' => true],
];
```

Then configure the bundle to your needs thanks to `config/packages/jane.yaml` file, for example:

```yaml
jane_auto_mapper:
  normalizer: false
  name_converter: ~
  cache_dir: '%kernel.cache_dir%/automapper'
  date_time_format: !php/const \DateTimeInterface::RFC3339_EXTENDED
  hot_reload: '%kernel.debug%'
```

Possible configuration fields:

- `normalizer` (default: `false`): A boolean which indicate if we inject the `AutoMapperNormalizer`;
- `name_converter` (default: `null`): A NameConverter based on your needs;
- `cache_dir` (default: `%kernel.cache_dir%/automapper`): This settings allows you to customize the output directory for generated mappers;
- `date_time_format` (default: `\DateTime::RFC3339`): This option allows you to change the date time format used to transform strings to `\DateTime`;
- `hot_reload` (default: `%kernel.debug%`): Will reload the AutoMapper registry every time you try to load new Mapper class, we recommend to put this option at `false` in production.

15.6 Extending the bundle

The AutoMapper comes with multiple elements to make it work, but you can custom many of them, this section will describe each of these customizable elements.

15.6.1 Mapper configuration

During automapping, we will build metadata about source & target data. Most of the time this process will be handled by one of our builtin extractor or the Symfony PropertyInfo component.

But you can customize this with a `MapperConfigurationInterface`. For example if you have an in input array as following:

```php
['name' => 'Jane Doe', 'age' => 25]
```

And we want to automap this array to an object that has a property *yearOfBirth*. With no configuration, this property will be skipped since there is no matching field in the array, but we can make a custom Mapper configuration to fill it.

We want to calculate this field based on the current year minus the `age` field. Here is a custom Mapper configuration definition following our example:

```php
use Jane\Bundle\AutoMapperBundle\Configuration\MapperConfigurationInterface;

class UserMapperConfiguration implements Configuration\MapperConfigurationInterface
{
    public function getSource(): string
    {
        return 'array';
    }

    public function getTarget(): string
    {
        return \Jane\Component\AutoMapper\Tests\Fixtures\UserDTO::class;
    }

    public function process(MapperGeneratorMetadataInterface $metadata): void
    {
        $metadata->forMember('yearOfBirth', function (array $user) {
            return ((int) date('Y')) - ((int) $user['age']);
        });
    }
}
```

This example will map the *yearOfBirth* field as stated!

If you are using the Bundle, any class implementing the `MapperConfigurationInterface` interface will be autoconfigured and linked to the AutoMapper instance.

15.6.2 Transformer

Sometimes we need to manage more complex objects that need specific behavior during mapping. For example the `Money\Money` object from the Money PHP library has a lot of properties we don't want to manage and can confuse the AutoMapper since it will try to map any properties.

For this kind of objects we need a custom TransformerFactory, you can see such a class in our test suite. You also need to implement the `TransformerFactoryInterface` interface in order to autoregister this factory in the AutoMapper.

15.6.3 NameConverter

As in Symfony, we have the possibility to overload the property names with NameConverter (see. related serializer documentation).

We can use the same behavior in the AutoMapper thanks to the `name_converter` configuration field. You have to give a service implementing the `AdvancedNameConverterInterface` interface.