

# The VarDumper Component

*The VarDumper component provides mechanisms for extracting the state out of any PHP variables. Built on top, it provides a better `dump()` function that you can use instead of [var\\_dump](#).*

## Installation

```
$ composer require --dev symfony/var-dumper
```

### Note

If you install this component outside of a Symfony application, you must require the `vendor/autoload.php` file in your code to enable the class autoloading mechanism provided by Composer. Read [this article](#) for more details.

### Note

If using it inside a Symfony application, make sure that the DebugBundle has been installed (or run `composer require --dev symfony/debug-bundle` to install it).

## The dump() Function

The VarDumper component creates a global `dump()` function that you can use instead of e.g. [var\\_dump](#). By using it, you'll gain:

- Per object and resource types specialized view to e.g. filter out Doctrine internals while dumping a single proxy entity, or get more insight on opened files with [stream\\_get\\_meta\\_data](#);
- Configurable output formats: HTML or colored command line output;
- Ability to dump internal references, either soft ones (objects or resources) or hard ones (`=&` on arrays or objects properties). Repeated occurrences of the same object/array/resource won't appear again and again anymore. Moreover, you'll be able to inspect the reference structure of your data;
- Ability to operate in the context of an output buffering handler.

For example:

```
require __DIR__.'/vendor/autoload.php';

// create a variable, which could be anything!
$someVar = ...;

dump($someVar);

// dump() returns the passed value, so you can dump an object and keep using it
dump($someObject)->someMethod();
```

By default, the output format and destination are selected based on your current PHP SAPI:

- On the command line (CLI SAPI), the output is written on STDOUT. This can be surprising to some because this bypasses PHP's output buffering mechanism;
- On other SAPIs, dumps are written as HTML in the regular output.

#### Tip

You can also select the output format explicitly defining the `VAR_DUMPER_FORMAT` environment variable and setting its value to either `html`, `cli` or [server](#).

#### Note

If you want to catch the dump output as a string, please read the [advanced section](#) which contains examples of it. You'll also learn how to change the format or redirect the output to wherever you want.

#### Tip

In order to have the `dump()` function always available when running any PHP code, you can install it globally on your computer:

1. Run `composer global require symfony/var-dumper`;
2. Add `auto_prepend_file = ${HOME}/.composer/vendor/autoload.php` to your `php.ini` file;
3. From time to time, run `composer global update symfony/var-dumper` to have the latest bug fixes.

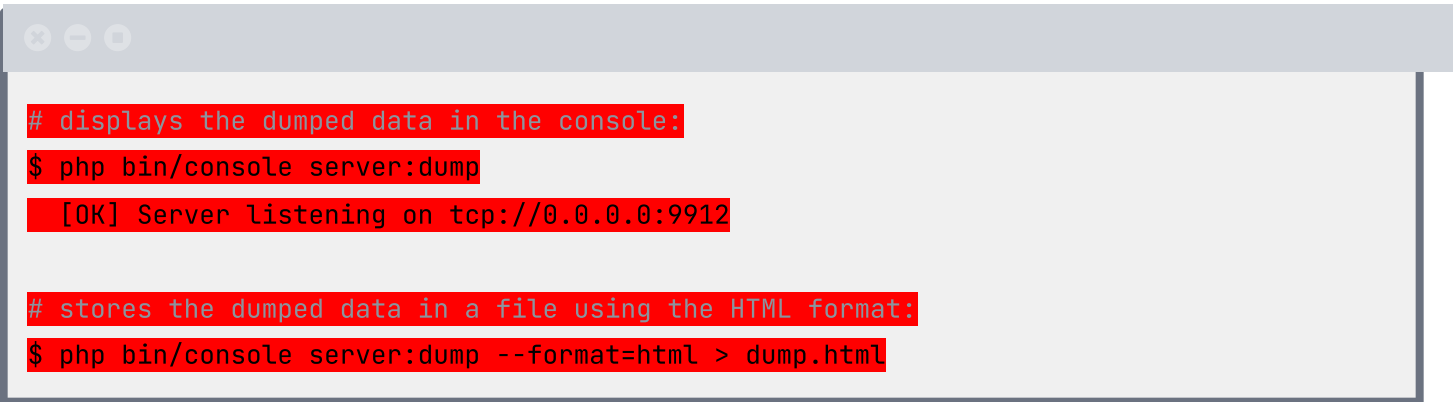
### Tip

The VarDumper component also provides a `dd()` ("dump and die") helper function. This function dumps the variables using `dump()` and immediately ends the execution of the script (using [exit](#)).

## The Dump Server

The `dump()` function outputs its contents in the same browser window or console terminal as your own application. Sometimes mixing the real output with the debug output can be confusing. That's why this component provides a server to collect all the dumped data.

Start the server with the `server:dump` command and whenever you call to `dump()`, the dumped data won't be displayed in the output but sent to that server, which outputs it to its own console or to an HTML file:



```
# displays the dumped data in the console:
$ php bin/console server:dump
[OK] Server listening on tcp://0.0.0.0:9912

# stores the dumped data in a file using the HTML format:
$ php bin/console server:dump --format=html > dump.html
```

Inside a Symfony application, the output of the dump server is configured with the [dump\\_destination option](#) of the debug package:

YAML | XML | PHP

```
# config/packages/debug.yaml
debug:
    dump_destination: "tcp://%env(VAR_DUMPER_SERVER)%"
```

Outside a Symfony application, use the [ServerDumper](#) class:

```
require __DIR__.'/vendor/autoload.php';

use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;
use Symfony\Component\VarDumper\Dumper\ContextProvider\CliContextProvider;
```

```

use Symfony\Component\VarDumper\Dumper\ContextProvider\SourceContextProvider;
use Symfony\Component\VarDumper\Dumper\HtmlDumper;
use Symfony\Component\VarDumper\Dumper\ServerDumper;
use Symfony\Component\VarDumper\VarDumper;

$cloner = new VarCloner();
$fallbackDumper = \in_array(\PHP_SAPI, ['cli', 'phpdbg']) ? new CliDumper() : new HtmlDumper;
$dumper = new ServerDumper('tcp://127.0.0.1:9912', $fallbackDumper, [
    'cli' => new CliContextProvider(),
    'source' => new SourceContextProvider(),
]);

VarDumper::setHandler(function (mixed $var) use ($cloner, $dumper): ?string {
    return $dumper->dump($cloner->cloneVar($var));
});

```

### Note

The second argument of [ServerDumper](#) is a [DataDumperInterface](#) instance used as a fallback when the server is unreachable. The third argument are the context providers, which allow to gather some info about the context in which the data was dumped. The built-in context providers are: cli, request and source.

Then you can use the following command to start a server out-of-the-box:

```

$ ./vendor/bin/var-dump-server
[OK] Server listening on tcp://127.0.0.1:9912

```

## Configuring the Dump Server with Environment Variables

If you prefer to not modify the application configuration (e.g. to quickly debug a project given to you) use the `VAR_DUMPER_FORMAT` env var.

First, start the server as usual:

```

$ ./vendor/bin/var-dump-server

```

Then, run your code with the `VAR_DUMPER_FORMAT=server` env var by configuring this value in the [.env file of your application](#). For console commands, you can also define this env var as follows:

```
$ VAR_DUMPER_FORMAT=server [your-cli-command]
```

#### Note

The host used by the server format is the one configured in the `VAR_DUMPER_SERVER` env var or `127.0.0.1:9912` if none is defined. If you prefer, you can also configure the host in the `VAR_DUMPER_FORMAT` env var like this: `VAR_DUMPER_FORMAT=tcp://127.0.0.1:1234`.

## DebugBundle and Twig Integration

The DebugBundle allows greater integration of this component into Symfony applications.

Since generating (even debug) output in the controller or in the model of your application may just break it by e.g. sending HTTP headers or corrupting your view, the bundle configures the `dump()` function so that variables are dumped in the web debug toolbar.

But if the toolbar cannot be displayed because you e.g. called `die()/exit()/dd()` or a fatal error occurred, then dumps are written on the regular output.

In a Twig template, two constructs are available for dumping a variable. Choosing between both is mostly a matter of personal taste, still:

- `{% dump foo.bar %}` is the way to go when the original template output shall not be modified: variables are not dumped inline, but in the web debug toolbar;
- on the contrary, `{{ dump(foo.bar) }}` dumps inline and thus may or not be suited to your use case (e.g. you shouldn't use it in an HTML attribute or a `<script>` tag).

This behavior can be changed by configuring the `debug.dump_destination` option. Read more about this and other options in [the DebugBundle configuration reference](#).

#### Tip

If the dumped contents are complex, consider using the local search box to look for specific variables or values. First, click anywhere on the dumped contents and then press `Ctrl. + F` or `Cmd. + F` to make the local search box appear. All the common shortcuts to navigate the search results are supported (`Ctrl. + G` or `Cmd. + G`, `F3`, etc.) When finished, press `Esc.` to hide the box again.

If you want to use your browser search input, press `Ctrl. + F` or `Cmd. + F` again while focusing on VarDumper's search input.

## Using the VarDumper Component in your PHPUnit Test Suite

The VarDumper component provides [a trait](#) that can help writing some of your tests for PHPUnit.

This will provide you with two new assertions:

### [assertDumpEquals\(\)](#)

verifies that the dump of the variable given as the second argument matches the expected dump provided as the first argument.

### [assertDumpMatchesFormat\(\)](#)

is like the previous method but accepts placeholders in the expected dump, based on the `assertStringMatchesFormat()` method provided by PHPUnit.

The `VarDumperTestTrait` also includes these other methods:

### [setUpVarDumper\(\)](#)

is used to configure the available casters and their options, which is a way to only control the fields you're expecting and allows writing concise tests.

### [tearDownVarDumper\(\)](#)

is called automatically after each case to reset the custom configuration made in `setUpVarDumper()`.

Example:

```
use PHPUnit\Framework\TestCase;
use Symfony\Component\VarDumper\Test\VarDumperTestTrait;

class ExampleTest extends TestCase
```

```

{
    use VarDumperTestTrait;

    protected function setUp(): void
    {
        $casters = [
            \DateTimeInterface::class => static function (\DateTimeInterface $date, array
                $stub->class = 'DateTime';
                return ['date' => $date->format('d/m/Y')]);
        ],
    ];

        $flags = CliDumper::DUMP_LIGHT_ARRAY | CliDumper::DUMP_COMMA_SEPARATOR;

        // this configures the casters & flags to use for all the tests in this class.
        // If you need custom configurations per test rather than for the whole class,
        // call this setUpVarDumper() method from those tests instead.
        $this->setUpVarDumper($casters, $flags);
    }

    public function testWithDumpEquals(): void
    {
        $testedVar = [123, 'foo'];

        // the expected dump contents don't have the default VarDumper structure
        // because of the custom casters and flags used in the test
        $expectedDump = <<<EOTXT
[
    123,
    "foo",
]
EOTXT;

        // if the first argument is a string, it must be the whole expected dump
        $this->assertDumpEquals($expectedDump, $testedVar);

        // if the first argument is not a string, assertDumpEquals() dumps it
        // and compares it with the dump of the second argument
        $this->assertDumpEquals($testedVar, $testedVar);
    }
}

```

# Dump Examples and Output

For simple variables, reading the output should be straightforward. Here are some examples showing first a variable defined in PHP, then its dump representation:

```
$var = [  
    'a simple string' => "in an array of 5 elements",  
    'a float' => 1.0,  
    'an integer' => 1,  
    'a boolean' => true,  
    'an empty array' => [],  
];  
dump($var);
```

```
array:5 [▼  
    "a simple string" => "in an array of 5 elements"  
    "a float" => 1.0  
    "an integer" => 1  
    "a boolean" => true  
    "an empty array" => []  
]
```

## Note

The gray arrow is a toggle button for hiding/showing children of nested structures.

```
$var = "This is a multi-line string.\n";  
$var .= "Hovering a string shows its length.\n";  
$var .= "The length of UTF-8 strings is counted in terms of UTF-8 characters.\n";  
$var .= "Non-UTF-8 strings length are counted in octet size.\n";  
$var .= "Because of this `\\xE9` octet (\\xE9),\n";  
$var .= "this string is not UTF-8 valid, thus the `b` prefix.\n";  
dump($var);
```

```
b""  
This is a multi-line string.  
Hovering a string shows its length.  
The length of UTF-8 strings is counted in terms of UTF-8 characters.  
Non-UTF-8 strings length are counted in octet size.  
Because of this `é` octet (xE9),  
this string is not UTF-8 valid, thus the `b` prefix.  
""
```



```

class PropertyExample
{
    public string $publicProperty = 'The `+` prefix denotes public properties,';
    protected string $protectedProperty = '`#` protected ones and `-` private ones.';
    private string $privateProperty = 'Hovering a property shows a reminder.';
}

$var = new PropertyExample();
dump($var);

```

```

PropertyExample {#14 ▼
  +publicProperty: "The `+` prefix denotes public properties,"
  #protectedProperty: "`#` protected ones and `-` private ones."
  -privateProperty: "Hovering a property shows a reminder."
}

```

### Note

`#14` is the internal object handle. It allows comparing two consecutive dumps of the same object.

```

class DynamicPropertyExample
{
    public string $declaredProperty = 'This property is declared in the class definition';
}

$var = new DynamicPropertyExample();
$var->undeclaredProperty = 'Runtime added dynamic properties have `` around their name.';
dump($var);

```

```

DynamicPropertyExample {#15 ▼
  +declaredProperty: "This property is declared in the class definition"
  +"undeclaredProperty": "Runtime added dynamic properties have `` around their name."
}

```

```

class ReferenceExample
{
    public string $info = "Circular and sibling references are displayed as `#number`.\nHo
}

$var = new ReferenceExample();

```

```
$var->aCircularReference = $var;
dump($var);
```

```
ReferenceExample {#16 ▼
  +info: ""
    Circular and sibling references are displayed as `#number`.
    Hovering them highlights all instances in the same dump.
    ""
  +"aCircularReference": ReferenceExample {#16}
}
```

```
$var = new \ErrorException(
    "For some objects, properties have special values\n"
    ."that are best represented as constants, like\n"
    ."`severity` below. Hovering displays the value (`2`).\n",
    0,
    E_WARNING
);
dump($var);
```

```
ErrorException {#14 ▼
  #message: ""
    For some objects, properties have special values
    that are best represented as constants, like
    `severity` below. Hovering displays the value (`2`).
    ""
  #code: 0
  #file: ".../var-dumper-example.php"
  #line: 58
  #severity: E_WARNING
  -trace: array:1 [▶]
}
```

```
$var = [];
$var[0] = 1;
$var[1] =& $var[0];
$var[1] += 1;
$var[2] = ["Hard references (circular or sibling)"];
$var[3] =& $var[2];
$var[3][] = "are dumped using `&number` prefixes.";
dump($var);
```

```
array:4 [▼
  0 => &1 2
  1 => &1 2
  2 => &2 array:2 [▼
    0 => "Hard references (circular or sibling)"
    1 => "are dumped using `&number` prefixes."
  ]
  3 => &2 array:2 [▶]
]
```

```
$var = new \ArrayObject();
$var[] = "Some resources and special objects like the current";
$var[] = "one are sometimes best represented using virtual";
$var[] = "properties that describe their internal state.";
dump($var);
```

```
ArrayObject {#17 ▼
  flag::STD_PROP_LIST: false
  flag::ARRAY_AS_PROPS: false
  iteratorClass: "ArrayIterator"
  storage: array:3 [▼
    0 => "Some resources and special objects like the current"
    1 => "one are sometimes best represented using virtual"
    2 => "properties that describe their internal state."
  ]
}
```

```
$var = new AcmeController(
    "When a dump goes over its maximum items limit,\n"
    ."or when some special objects are encountered,\n"
    ."children can be replaced by an ellipsis and\n"
    ."optionally followed by a number that says how\n"
    ."many have been removed; `9` in this case.\n"
);
dump($var);
```

```
AcmeController {#14 ▼
  -info: ""
    When a dump goes over its maximum items limit,
    or when some special objects are encountered,
    children can be replaced by an ellipsis and
    optionnally followed by a number that says how
    many have been removed; `9` in this case.
    ""
  #container: Symfony\Component\DependencyInjection\Container {#15 ...9}
}
```

```

class Foo
{
    // $foo is uninitialized, which is different from being null
    private int|float $foo;
    public ?string $baz = null;
}

$var = new Foo();
dump($var);

```

```

App\Service\Foo {#1602 ▼
  -foo: ? int|float
  +baz: null
}

```

## Advanced Usage

The `dump()` function is just a thin wrapper and a more convenient way to call `VarDumper::dump()`. You can change the behavior of this function by calling `VarDumper::setHandler($callable)`. Calls to `dump()` will then be forwarded to `$callable`.

By adding a handler, you can customize the [Cloners](#), [Dumpers](#) and [Casters](#) as explained below. A simple implementation of a handler function might look like this:

```

use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;
use Symfony\Component\VarDumper\Dumper\HtmlDumper;
use Symfony\Component\VarDumper\VarDumper;

VarDumper::setHandler(function (mixed $var): ?string {
    $cloner = new VarCloner();
    $dumper = 'cli' === PHP_SAPI ? new CliDumper() : new HtmlDumper();

    return $dumper->dump($cloner->cloneVar($var));
});

```

## Cloners

A cloner is used to create an intermediate representation of any PHP variable. Its output is a [Data](#) object that wraps this representation.

You can create a Data object this way:

```
use Symfony\Component\VarDumper\Cloner\VarCloner;

$cloner = new VarCloner();
$data = $cloner->cloneVar($myVar);
// this is commonly then passed to the dumper
// see the example at the top of this page
// $dumper->dump($data);
```

Whatever the cloned data structure, resulting Data objects are always serializable.

A cloner applies limits when creating the representation, so that one can represent only a subset of the cloned variable. Before calling [cloneVar\(\)](#), you can configure these limits:

#### [setMaxItems\(\)](#)

Configures the maximum number of items that will be cloned *past the minimum nesting depth*. Items are counted using a breadth-first algorithm so that lower level items have higher priority than deeply nested items. Specifying -1 removes the limit.

#### [setMinDepth\(\)](#)

Configures the minimum tree depth where we are guaranteed to clone all the items. After this depth is reached, only `setMaxItems` items will be cloned. The default value is 1, which is consistent with older Symfony versions.

#### [setMaxString\(\)](#)

Configures the maximum number of characters that will be cloned before cutting overlong strings. Specifying -1 removes the limit.

Before dumping it, you can further limit the resulting [Data](#) object using the following methods:

#### [withMaxDepth\(\)](#)

Limits dumps in the depth dimension.

#### [withMaxItemsPerDepth\(\)](#)

Limits the number of items per depth level.

#### [withRefHandles\(\)](#)

Removes internal objects' handles for sparser output (useful for tests).

#### [seek\(\)](#)

Selects only sub-parts of already cloned arrays, objects or resources.

Unlike the previous limits on cloners that remove data on purpose, these can be changed back and forth before dumping since they do not affect the intermediate representation internally.

#### Note

When no limit is applied, a [Data](#) object is as accurate as the native [serialize](#) function, and thus could be used for purposes beyond debugging.

## Dumpers

A dumper is responsible for outputting a string representation of a PHP variable, using a [Data](#) object as input. The destination and the formatting of this output vary with dumpers.

This component comes with an [HtmlDumper](#) for HTML output and a [CliDumper](#) for optionally colored command line output.

For example, if you want to dump some `$variable`, do:

```
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$cloner = new VarCloner();
$dumper = new CliDumper();

$dumper->dump($cloner->cloneVar($variable));
```

By using the first argument of the constructor, you can select the output stream where the dump will be written. By default, the `CliDumper` writes on `php://stdout` and the `HtmlDumper` on `php://output`. But any PHP stream (resource or URL) is acceptable.

Instead of a stream destination, you can also pass it a callable that will be called repeatedly for each line generated by a dumper. This callable can be configured using the first argument of a dumper's constructor, but also using the [setOutput\(\)](#) method or the second argument of the [dump\(\)](#) method.

For example, to get a dump as a string in a variable, you can do:

```
use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;
```

```

$cloner = new VarCloner();
$dumper = new CliDumper();
$output = '';

$dumper->dump(
    $cloner->cloneVar($variable),
    function (int $line, int $depth) use (&$output): void {
        // A negative depth means "end of dump"
        if ($depth >= 0) {
            // Adds a two spaces indentation to the line
            $output .= str_repeat(' ', $depth).$line."\n";
        }
    }
);

// $output is now populated with the dump representation of $variable

```

Another option for doing the same could be:

```

use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$cloner = new VarCloner();
$dumper = new CliDumper();
$output = fopen('php://memory', 'r+b');

$dumper->dump($cloner->cloneVar($variable), $output);
$output = stream_get_contents($output, -1, 0);

// $output is now populated with the dump representation of $variable

```

### Tip

You can pass true to the second argument of the [dump\(\)](#) method to make it return the dump as a string:

```

$output = $dumper->dump($cloner->cloneVar($variable), true);

```

Dumpers implement the [DataDumperInterface](#) interface that specifies the [dump\(Data \\$data\)](#) method. They also typically implement the [DumperInterface](#) that frees them from re-implementing the logic required to walk through a [Data](#) object's internal structure.

The [HtmlDumper](#) uses a dark theme by default. Use the [setTheme\(\)](#) method to use a light theme:

```
// ...  
$htmlDumper->setTheme('light');
```

The [HtmlDumper](#) limits string length and nesting depth of the output to make it more readable. These options can be overridden by the third optional parameter of the [dump\(Data \\$data\)](#) method:

```
use Symfony\Component\VarDumper\Dumper\HtmlDumper;  
  
$output = fopen('php://memory', 'r+b');  
  
$dumper = new HtmlDumper();  
$dumper->dump($var, $output, [  
    // 1 and 160 are the default values for these options  
    'maxDepth' => 1,  
    'maxLength' => 160,  
]);
```

The output format of a dumper can be fine tuned by the two flags `DUMP_STRING_LENGTH` and `DUMP_LIGHT_ARRAY` which are passed as a bitmap in the third constructor argument. They can also be set via environment variables when using [assertDumpEquals\(\\$dump, \\$data, \\$filter, \\$message\)](#) during unit testing.

The `$filter` argument of `assertDumpEquals()` can be used to pass a bit field of `Caster::EXCLUDE_*` constants and influences the expected output produced by the different casters.

If `DUMP_STRING_LENGTH` is set, then the length of a string is displayed next to its content:

```
use Symfony\Component\VarDumper\Cloner\VarCloner;  
use Symfony\Component\VarDumper\Dumper\AbstractDumper;  
use Symfony\Component\VarDumper\Dumper\CliDumper;  
  
$varCloner = new VarCloner();  
$var = ['test'];
```



```

$dumper = new CliDumper();
echo $dumper->dump($varCloner->cloneVar($var), true);

// array:1 [
//    0 => "test"
// ]

$dumper = new CliDumper(null, null, AbstractDumper::DUMP_STRING_LENGTH);
echo $dumper->dump($varCloner->cloneVar($var), true);

// (added string length before the string)
// array:1 [
//    0 => (4) "test"
// ]

```

If `DUMP_LIGHT_ARRAY` is set, then arrays are dumped in a shortened format similar to PHP's short array notation:

```

use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\AbstractDumper;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$varCloner = new VarCloner();
$var = ['test'];

$dumper = new CliDumper();
echo $dumper->dump($varCloner->cloneVar($var), true);

// array:1 [
//    0 => "test"
// ]

$dumper = new CliDumper(null, null, AbstractDumper::DUMP_LIGHT_ARRAY);
echo $dumper->dump($varCloner->cloneVar($var), true);

// (no more array:1 prefix)
// [
//    0 => "test"
// ]

```

If you would like to use both options, then you can combine them by using the logical OR operator `|`:

```

use Symfony\Component\VarDumper\Cloner\VarCloner;
use Symfony\Component\VarDumper\Dumper\AbstractDumper;
use Symfony\Component\VarDumper\Dumper\CliDumper;

$varCloner = new VarCloner();
$var = ['test'];

$dumper = new CliDumper(null, null, AbstractDumper::DUMP_STRING_LENGTH | AbstractDumper::DUMP_VERBOSE, AbstractDumper::DUMP_FORMAT);
echo $dumper->dump($varCloner->cloneVar($var), true);

// [
//   0 => (4) "test"
// ]

```

## Casters

Objects and resources nested in a PHP variable are "cast" to arrays in the intermediate [Data](#) representation. You can customize the array representation for each object/resource by hooking a Caster into this process. The component already includes many casters for base PHP classes and other common classes.

If you want to build your own Caster, you can register one before cloning a PHP variable. Casters are registered using either a Cloner's constructor or its `addCasters()` method:

```

use Symfony\Component\VarDumper\Cloner\VarCloner;

$myCasters = [...];
$cloner = new VarCloner($myCasters);

// or

$cloner->addCasters($myCasters);

```

The provided `$myCasters` argument is an array that maps a class, an interface or a resource type to a callable:

```

$myCasters = [
    'FooClass' => $myFooClassCallableCaster,
    ':bar resource' => $myBarResourceCallableCaster,
];

```

As you can notice, resource types are prefixed by a `:` to prevent colliding with a class name.

Because an object has one main class and potentially many parent classes or interfaces, many casters can be applied to one object. In this case, casters are called one after the other, starting from casters bound to the interfaces, the parents classes and then the main class. Several casters can also be registered for the same resource type/class/interface. They are called in registration order.

Casters are responsible for returning the properties of the object or resource being cloned in an array. They are callables that accept five arguments:

- the object or resource being casted;
- an array modeled for objects after PHP's native `(array)` cast operator;
- a [Stub](#) object representing the main properties of the object (class, type, etc.);
- `true/false` when the caster is called nested in a structure or not;
- A bit field of [Caster](#) `::EXCLUDE_*` constants.

Here is a simple caster not doing anything:

```
use Symfony\Component\VarDumper\Cloner\Stub;

function myCaster(mixed $object, array $array, Stub $stub, bool $isNested, int $filter): array
{
    // ... populate/alter $array to your needs

    return $array;
}
```

For objects, the `$array` parameter comes pre-populated using PHP's native `(array)` casting operator or with the return value of `$object->__debugInfo()` if the magic method exists. Then, the return value of one Caster is given as the array argument to the next Caster in the chain.

When casting with the `(array)` operator, PHP prefixes protected properties with a `\0*\0` and private ones with the class owning the property. For example, `\0Foobar\0` will be the prefix for all private properties of objects of type Foobar. Casters follow this convention and add two more prefixes: `\0~\0` is used for virtual properties and `\0+\0` for dynamic ones (runtime added properties not in the class declaration).

#### Note

Although you can, it is advised to not alter the state of an object while casting it in a Caster.

### Tip

Before writing your own casters, you should check the existing ones.

## Adding Semantics with Metadata

Since casters are hooked on specific classes or interfaces, they know about the objects they manipulate. By altering the `$stub` object (the third argument of any caster), one can transfer this knowledge to the resulting `Data` object, thus to dumpers. To help you do this (see the source code for how it works), the component comes with a set of wrappers for common additional semantics. You can use:

- [ConstStub](#) to wrap a value that is best represented by a PHP constant;
- [ClassStub](#) to wrap a PHP identifier (*i.e.* a class name, a method name, an interface, *etc.*);
- [CutStub](#) to replace big noisy objects/strings/*etc.* by ellipses;
- [CutArrayStub](#) to keep only some useful keys of an array;
- [ImgStub](#) to wrap an image;
- [EnumStub](#) to wrap a set of virtual values (*i.e.* values that do not exist as properties in the original PHP data structure, but are worth listing alongside with real ones);
- [LinkStub](#) to wrap strings that can be turned into links by dumpers;
- [TraceStub](#) and their
- [FrameStub](#) and
- [ArgsStub](#) relatives to wrap PHP traces (used by [ExceptionCaster](#)).

For example, if you know that your `Product` objects have a `brochure` property that holds a file name or a URL, you can wrap them in a `LinkStub` to tell `HtmlDumper` to make them clickable:

```
use Symfony\Component\VarDumper\Caster\LinkStub;
use Symfony\Component\VarDumper\Cloner\Stub;

function ProductCaster(Product $object, array $array, Stub $stub, bool $isNested, int $fil
{
    $array['brochure'] = new LinkStub($array['brochure']);
```

```
return $array;  
}
```

This work, including the code samples, is licensed under a [Creative Commons BY-SA 3.0](#) license.

Symfony™ is a trademark of Symfony SAS. All rights reserved.