BeBat/Verify

Release latest

Ben Batschelet

Aug 18, 2023

CONTENTS

BeBat/Verify is a small wrapper for PHPUnit's assertions, intended to make your assertion code cleaner, easier to understand, and simpler to maintain. Here you will find all the information needed to dive into BeBat/Verify and start using it in your testing journey.

GETTING STARTED

1.1 Installation

To install the current version of BeBat/Verify from Packagist, run the following in your project directory:

```
composer require --dev bebat/verify
```

BeBat/Verify will be added to your `composer.json` under `require-dev` and installed in your `vendor` directory.

1.1.1 Compatibility

BeBat/Verify is built on top of PHPUnit's own assertions. It is compatible with any version of PHPUnit 8, 9, or 10.1 and above. It should also be compatible with the current version of Codeception.

Some assertions have been removed from later versions of PHPUnit, and others added. When using BeBat/Verify you should explicitly declare what major version of PHPUnit your project depends on so that there are no surprise compatibility issues. See the *available assertions* to see what assertions are compatible with your version of PHPUnit.

In addition, BeBat/Verify is compatible with both PHP 7.2+ and 8+.

1.2 Basic Usage

BeBat/Verify uses namespaced functions, so to include it in your unit tests you should add a `use function` statement to the top of your test files:

```php
// assertions for values in code
use function BeBat\Verify\verify;

// assertions for files
use function BeBat\Verify\verify_file;
```

To use BeBat/Verify in your tests, pass the subject to `verify()`, followed by a *conjunction*, and then your *assertion(s)*. For example:

```php
$testValue = true;

verify($testValue)->is()->true();
```

That's it! You've now asserted that `$testValue` is `true`!

1.2.1 Alternate Functions

To better match TDD/BDD style, you may wish to give BeBat/Verify's functions a different name like `expect()`. This can be done through the use of function aliases like so:

```php
use function BeBat\Verify\verify as expect;
use function BeBat\Verify\verify_file as expect_file;
```

Now, in your unit test code, you can write:

```php
expect($testValue)->will()->be()->equalTo('some other value');
```

CONJUNCTIONS

Conjunctions are used tie your *subject* to your *assertions*. They control whether the *assertion* is "positive" (ie, assert that subject *is* a certain value) or "negative" (subject *is not* a certain value). There are also "neutral" conjunctions that do not change whether the assertion is positive or negative; they can be used to make your tests more readable. The default set of conjunctions are:

Positive

- `is()`
- `will()`
- `does()`
- `has()`

Negative

- `isNot()`
- `willNot()`
- `doesNot()`

Neutral

- `and()`
- `be()`
- `have()`

2.1 Descriptions

Conjunctions also allow you to pass a description of your assertion to BeBat/Verify. If your assertion fails, PHPUnit will use this description as the failure message. For example:

```
verify($myObject->isValid())->will('pass validation')->be()->true();
```

If this assertion fails, you will see `Value will pass validation` in PHPUnit's output. BeBat/Verify uses a generic term by default (`Value`), as well as the conjunction to create the full description. If you would like to use a more descriptive name for your subject you can pass that to `verify()` as well. For example:

```
verify($gpa, 'Student GPA')->isNot('failing')->lessThan(2.0);
```

The description in this case would be `Student GPA is not failing`.

2.2 Custom Conjunctions

Conjunctions are configured through a set of static arrays of strings in `BeBat\Verify\API\Base`. This allows you to further customize the description messages, as well as tailor the conjunctions to your own writing style. You can manipulate these value like you would any other array. For example:

```
BeBat\Verify\API\Base::$positiveConjunctions[] = 'to';
BeBat\Verify\API\Base::$positiveConjunctions[] = 'should';

BeBat\Verify\API\Base::$negativeConjunctions[] = 'shouldNot';

BeBat\Verify\API\Base::$neutralConjunctions[] = 'also';
BeBat\Verify\API\Base::$neutralConjunctions[] = 'or';
```

This should be performed somewhere in your test suite's bootstrap code so that it is done before any assertions are called and is shared across your tests.

ASSERTIONS

This page lists all of the assertions built in to BeBat/Verify.

Important: All of the examples on this page use positive conjunctions. If you want to verify the inverse of any of these assertions you should use a *negative conjunction* instead.

3.1 Value Assertions

To make assertions about the value of some entity you should pass it to `verify()` and then chain your assertion(s) after a conjunction.

3.1.1 Equality

identicalTo()

Listing 1: Assert that subject has the same type and value as some other entity

```
verify($subject)->is()->identicalTo('some value');
```

equalTo()

Listing 2: Assert that subject has the same value as some other entity

```
verify($subject)->is()->equalTo('some value');
```

Note: The behavior of equalTo() can be changed using the *within()*, *withoutOrder()*, *withoutCase()*, and *withoutLinEndings()* modifiers.

equalToFile()

Listing 3: Assert that subject has the same value as the contents of a file

```
verify($subject)->is()->equalToFile('/path/to/file.txt');
```

Note: The behavior of equalToFile() can be changed using the *withoutCase()* and *withoutLineEndings()* modifiers.

3.1.2 Truthiness

true()

Listing 4: Assert that subject is true

```
verify($subject)->is()->true();
```

false()

Listing 5: Assert that subject is false

```
verify($subject)->is()->false();
```

null()

Listing 6: Assert that subject is null

```
verify($subject)->is()->null();
```

empty()

Listing 7: Assert that subject is empty

```
verify($subject)->is()->empty();
```

passCallback()

Listing 8: Assert that subject will pass a callback function

```
verify($subject)->will()->passCallback(function ($value): bool {
    return isPrime($value);
});
```

3.1.3 Type

instanceOf()

Listing 9: Assert that subject is an instance of some class

```
verify($subject)->is()->instanceOf(MyClass::class);
```

array()

Listing 10: Assert that subject is an array

```
verify($subject)->is()->array();
```

bool()

Listing 11: Assert that subject is a boolean

```
verify($subject)->is()->bool();
```

callable()

Listing 12: Assert that subject is callable

```
verify($subject)->is()->callable();
```

closed()

Listing 13: Assert that subject is a closed resource

```
verify($subject)->is()->closed();
```

> **Attention:** The `closed()` assertion requires PHPUnit 9 or later.

float()

Listing 14: Assert that subject is a floating point number

```
verify($subject)->is()->float();
```

int()

Listing 15: Assert that subject is an integer number

```
verify($subject)->is()->int();
```

iterable()

Listing 16: Assert that subject is an iterable type

```
verify($subject)->is()->iterable();
```

numeric()

Listing 17: Assert that subject is a numeric type

```
verify($subject)->is()->numeric();
```

object()

Listing 18: Assert that subject is an object

```
verify($subject)->is()->object();
```

resource()

Listing 19: Assert that subject is a resource

```
verify($subject)->is()->resource();
```

scalar()

Listing 20: Assert that subject is a scalar value

```
verify($subject)->is()->scalar();
```

string()

Listing 21: Assert that subject is a string

```
verify($subject)->is()->string();
```

3.1.4 Numeric Values

lessThan()

Listing 22: Assert that subject is less than some value

```
verify($subject)->is()->lessThan($value);
```

lessOrEqualTo()

Listing 23: Assert that subject is less than or equal to some value

```
verify($subject)->is()->lessOrEqualTo($value);
```

greaterThan()

Listing 24: Assert that subject is greater than some value

```
verify($subject)->is()->greaterThan($value);
```

greaterOrEqualTo()

Listing 25: Assert that subject is greater than or equal to some value

```
verify($subject)->is()->greaterOrEqualTo($value);
```

finite()

Listing 26: Assert that subject is a finite value

```
verify($subject)->is()->finite();
```

infinite()

Listing 27: Assert that subject is an infinite value

```
verify($subject)->is()->infinite();
```

nan()

Listing 28: Assert that subject is a NaN (or "not a number") value

```
verify($subject)->is()->nan();
```

3.1.5 String Values

contain()

Listing 29: Assert that subject contains a value

```
verify($subject)->will()->contain('value');
```

Note: The behavior of `contain()` can be changed using the *withoutCase()* and *withoutLinEndings()* modifiers.

startWith()

Listing 30: Assert that subject starts with some value

```
verify($subject)->wil()->startWith('value');
```

endWith()

Listing 31: Assert that subject ends with some value

```
verify($subject)->will()->endWith('value');
```

matchRegExp()

Listing 32: Assert that subject matches a regular expression

```
verify($subject)->will()->matchRegExp('/myregexp/');
```

matchFormat()

Listing 33: Assert that subject matches a format pattern

```
verify($subject)->will()->matchFormat('%i');
```

See also:

See PHPUnit's assertStringMatchesFormat() for details on format placeholders.

matchFormatFile()

Listing 34: Assert that subject matches a format pattern from a file

```
verify($subject)->will()->matchFormatFile('/path/to/format.txt');
```

3.1.6 Array Values

contain()

Listing 35: Assert that subject contains some value

```
verify($subject)->will()->contain('value');
```

Note: Unlike PHP and PHPUnit, BeBat/Verify's `contain()` performs *strict* comparison by default for both objects and internal types. If your test(s) require loose type checking you must use a *modifier*.

Note: The behavior of `contain()` can be changed using the *withoutType()* and *withoutIdentity()* modifiers.

key()

Listing 36: Assert that subject has a given key

```
verify($subject)->has()->key('value');
```

count()

Listing 37: Assert that subject has a certain number of elements

```
verify($subject)->has()->count(4);
```

sameSizeAs()

Listing 38: Assert that subject has the same number of elements as another array or traversable value

```
verify($subject)->is()->sameSizeAs(['some', 'array']);
```

containOnly()

Listing 39: Assert that subject only contains values of a given type

```
verify($subject)->will()->containOnly('string');
verify($subject)->will()->containOnly(SomeClass::class);
```

Note: The containOnly() assertion works for both internal types and classes.

list()

Listing 40: Assert that subject is a list (all keys are consecutive numbers starting at 0)

```
verify($subject)->is()->list();
```

Attention: The list() assertion requires PHPUnit 10 or later.

3.1.7 Object & Class Properties

attribute()

Listing 41: Assert that subject has some attribute/property

```
verify($subject)->has()->attribute('attributeName');
verify(MyClass::class)->has()->attribute('attributeName');
```

Deprecated since version 3.2.0: The attribute() assertion has been replaced by *property()*

property()

Listing 42: Assert that subject has some property

```
verify($subject)->has()->property('propertyName');
```

Deprecated since version 3.2.0: Using the property() assertion with a class string as the subject has been deprecated. Assertions with object instances as the subject will continue to be supported.

staticAttribute()

Listing 43: Assert that subject has a static attribute/property

```
verify(MyClass::class)->has()->staticAttribute('attributeName');
```

Deprecated since version 3.2.0: Making assertions about static attributes has been deprecated.

3.1.8 JSON

json()

Listing 44: Assert that subject is a valid JSON string

```
verify($subject)->is()->json();
```

equalToJsonString()

Listing 45: Assert that subject is equal to a JSON string

```
verify($subject)->is()->equalToJsonString('{"json": "string"}');
```

equalToJsonFile()

Listing 46: Assert that subject is equal to a JSON value from a file

```
verify($subject)->is()->equalToJsonFile('/path/to/file.json');
```

3.2 File Assertions

BeBat/Verify includes assertions specific to filesystem entries. To make assertions about a filesystem entity, pass the path to `verify_file()` and then chain your assertion(s) after a conjunction.

3.2.1 State & Type

exist()

Listing 47: Assert that a path exists in the filesystem

```
verify_file($path)->does()->exist();
```

file()

Listing 48: Assert that a path is a regular file

```
verify_file($path)->is()->file();
```

directory()

Listing 49: Assert that a path is a directory

```
verify_file($path)->is()->directory();
```

link()

Listing 50: Assert that a path is a symbolic link

```
verify_file($path)->is()->link();
```

3.2.2 Contents & Equality

equalTo()

Listing 51: Assert that the file's contents are equal to some string

```
verify_file($file)->is()->equalTo('value');
```

Note: The behavior of equalTo() can be changed using the *withoutCase()* and *withoutLinEndings()* modifiers.

equalToFile()

Listing 52: Assert that the file's contents are equal to another file's

```
verify_file($file)->is()->equalToFile('/path/to/file.txt');
```

Note: The behavior of equalToFile() can be changed using the *withoutCase()* and *withoutLineEndings()* modifiers.

contain()

Listing 53: Assert that the file's contents contains some value

```
verify_file($file)->will()->contain('value');
```

Note: The behavior of `contain()` can be changed using the *withoutCase()* and *withoutLinEndings()* modifiers.

containFiles()

Listing 54: Assert that a directory contains some files

```
verify_file($directory)->will()->containFiles(['File1', 'File2']);
```

linkTarget()

Listing 55: Assert that a symbolic link points to a particular file

```
verify_file($link)->has()->linkTarget('/some/other/file');
```

passCallback()

Listing 56: Assert that the file's contents will pass a callback function

```
verify_file($file)->will()->passCallback(function($content): bool {
    return complexValidationChecks($content);
});
```

3.2.3 Permissions

readable()

Listing 57: Assert that a file is readable

```
verify_file($file)->is->readable();
```

writable()

Listing 58: Assert that a file is writable

```
verify_file($file)->is()->writable()
```

executable()

Listing 59: Assert that a file is executable

```
verify_file($file)->is()->executable()
```

permission()

Listing 60: Assert that a file has some permission value

```
verify_file($file)->has()->permission(0755);
verify_file($file)->has()->permission('644');
```

Note: The `permission()` assertion accepts permissions in octal format as either strings or integers.

Note: The behavior of `permission()` can be changed use the *matching()* modifier.

3.2.4 Ownership

owner()

Listing 61: Assert that a file is owned by a given user

```
verify_file($file)->has()->owner(501);
verify_file($file)->has()->owner('username');
```

Note: The `owner()` assertion supports both user names and IDs.

group()

Listing 62: Assert that a file belongs to a given group

```
verify_file($file)->has()->group(1001);
verify_file($file)->has()->group('groupname');
```

Note: The group() assertion supports both group names and IDs.

3.2.5 JSON

json()

Listing 63: Assert that the contents of a file are valid JSON

```
verify_file($file)->is()->json();
```

equalToJsonString()

Listing 64: Assert that the contents of a file are equal to a given JSON string

```
verify_file($file)->is()->equalToJsonString('{"json": "string"');
```

equalToJsonFile()

Listing 65: Assert that the contents of a file are equal to a different JSON file

```
verify_file($file)->is()->equalToJsonFile('/path/to/file.json');
```

FOUR

ASSERTION MODIFIERS

The behavior of many assertions can be adjusted inline with the test. These modifiers can be used to control case sensitivity, account for floating point errors, or strictness when checking for object identity and datatypes.

4.1 Included Modifiers

4.1.1 `within()`

Listing 1: Account for floating point errors

```
verify(0.1 + 0.2)->within(0.01)->is()->equalTo(0.3);
```

Supported Assertion

- verify(<float>)
    - *equalTo()*

4.1.2 `withoutCase()`

Listing 2: Ignore case when comparing strings

```
verify('A String')->withoutCase()->is()->equalTo('a string');
verify('A String')->withoutCase()->will()->contain('string');
verify('a string')->withoutCase()->is()->equalToFile('/some/file.txt');

verify_file('/some/file.txt')->withoutCase()->is()->equalTo('a string');
verify_file('/some/file.txt')->withoutCase()->will()->contain('string');
verify_file('/some/file.txt')->withoutCase()->is()->equalToFile('/some/other/file.txt');
```

Supported Assertions

- verify(<string>)
    - *equalTo()*
    - *contain()*
    - *equalToFile()*
- verify_file()
    - *equalTo()*
    - *contain()*
    - *equalToFile()*

4.1.3 withoutLineEndings()

Listing 3: Ignore line ending format when comparing strings

```
verify("a\nstring")->withoutLineEndings()->is()->equalTo("a\r\nstring");
verify("another\nstring")->withoutLineEndings()->will()->contain("other\r\nstring");
verify("a\r\nstring")->withoutLineEndings()->is()->equalToFile('/some/file.txt');

verify_file('/some/file.txt')->withoutLineEndings()->is()->equalTo("a\r\nstring")
verify_file('/some/file.txt')->withoutLineEndings()->will()->contain("other\r\nstring")
verify_file('/some/file.txt')->withoutLineEndings()->is()->equalToFile('/some/other/file.
↪txt')
```

Attention: The `withoutLineEndings()` modifier requires PHPUnit 10 or later.

Supported Assertions

- `verify(<string>)`
    - *equalTo()*
    - *contain()*
    - *equalToFile()*
- `verify_file()`
    - *equalTo()*
    - *contain()*
    - *equalToFile()*

4.1.4 `withoutOrder()`

Listing 4: Ignore element ordering when comparing arrays

```
verify([1, 2, 3])->withoutOrder()->is()->equalTo([3, 1, 2]);
```

Supported Assertion

- verify(<array>)
    - *equalTo()*

4.1.5 withoutIdentity()

Listing 5: Ignore object identity when comparing values

```
verify([$objectA])->withoutIdentity()->does()->contain($objectB);
```

Supported Assertion

- verify(<array>)
    - *contain()*

4.1.6 withoutType()

Listing 6: Ignore data type when comparing values

```
verify(['1', '2'])->withoutType()->will()->contain(1);
```

Supported Assertion

- verify(<array>)
    - *contain()*

4.1.7 matching()

Listing 7: Match a minimum set of permissions, rather than an exact value

```
verify_file('/some/file/to/test.sh')->has()->matching()->permissions(0711);
```

Supported Assertion

- verify_file()
  - *permission()*

4.2 Chaining Modifiers

Modifiers can be *chained* inline with an assertion, so any assertion that supports both *withoutCase()* and *withoutLineEndings()* will support applying both modifiers simultaneously.

```
verify("A\n\rString")->withoutCase()->withoutLineEndings()
    ->is()->equalTo("a\nstring");
```

BeBat/Verify resets its internal state after each assertion, so if you are chaining modifiers along with multiple assertions, you must reapply the modifier each time.

```
verify(['1', '2', '3'])->will()->withoutType()->contain(1)
    ->and()->withoutType()->contain(2);
```

CHAINING

Multiple *conjunctions* and *assertions* can be chained together, allowing developers to write multiple assertions about one subject very easily. For example:

```
verify($value)->is()->internalType('array')
    ->and()->has()->key('my_index')
    ->and()->will()->contain('my value');
```

The above performs three separate assertions against `$value` in sequence, without having to redeclare our subject, and does so in a concise, easy to read syntax.

You can switch between positive and negative assertions on the fly; the condition will apply to whatever assertions follow it. For example:

```
verify($value)->will()->contain('value 1')
    ->and()->contain('value 2')
    ->and()->doesNot()->contain('value c')
    ->and()->doesNot()->contain('value d');
```

The above snippet will assert that `$value` contains `'value 1'` and `'value 2'`, and does *not* contain `'value c'` or `'value d'`. It is worth noting that BeBat/Verify requires just a single positive or negative conjunction; any additional conjunction that does not change the assertion condition is optional. So the previous example could be simplified to:

```
verify($value)->will()->contain('value 1')->contain('value 2')
    ->doesNot()->contain('value c')->contain('value d');
```

Additional conjunctions are *only* required if you are changing to a positive or negative condition for the following assertion(s), or if you wish to add a descriptive message to the assertion:

```
verify($starsArray, 'Famous People')->will('have a Beatle')->contain('Ringo')
    ->will('have a cartoon')->contain('Bugs Bunny')
    ->willNot('have a pirate')->contain('Stede Bonnet');
```

PROPERTY ASSERTIONS

BeBat/Verify has the ability to test the value of object and class properties, even those that are protected or private. While writing assertions about a subject's internal state is not generally good practice, there are times when inspecting a protected value may be the simplest way of checking your code. The property you wish to check can be tacked on after calling `verify()`, just like if you were accessing it as a public value.

For example, if you had an object called `$user` with a `first_name` property that should be equal to `'Alice'`, you can assert that with the following code:

```
verify($user)->first_name->is()->equalTo('Alice');
```

A similar assertion about a class's static properties might look like the following:

```
verify(Model::class)->dbc->is()->resource();
```

If you would rather explicitly identify your property, you can do so with the `propertyNamed()` method:

```
verify($obj)->propertyNamed('fooBar')->is()->false();
```

All of BeBat/Verify's assertions should be compatible with reading object or class properties. In addition, properties fully support chaining and assertion modifiers. The only exception is that once your chain contains an attribute, you can no longer add assertions about their containing object. Put another way, always write your assertions about an object *first* before writing any about its properties. For example:

```
verify($model)->isNot()->null()
    ->and()->is()->instanceOf(MyModelClass::class)
    ->and()->first_name->withoutCase()->is()->equalTo('sally')
    ->and()->last_name->withoutCase()->doesNot()->contain('smith')
    ->and()->gpa->within(0.01)->is()->equalTo(4.0);
```

METHOD ASSERTIONS

Just like with *properties*, assertions can be made about an object's methods by adding the method call after `verify()`. You can write assertions about either a method's return value or an exception that the method throws.

7.1 Return Values

A simple example might look like:

```
verify($calculator)->add(2, 3)->will()->returnValue()->identicalTo(5);
```

In `returnValue()`, BeBat/Verify will call `add()` on the `$calculator` object, passing it 2 and 3, and then cache its result internally. This means you can write multiple assertions about the return value, just like other verifiers, without the method needing to be called again.

If your method name conflicts with part of the verifier API, you can use `method()` and `with()` to explicitly set a method name and arguments:

```
verify(new ArrayObject([]))
    ->method('empty')->will()->returnValue()->true()
    ->method('count')->will()->returnValue()->identicalTo(0);
```

The `with()` method can also be used to set up multiple example arguments for a single method:

```
verify($calculator)->add()
    ->with(1, $someValue)->will()->returnValue()->greaterThan($someValue)
    ->with(0, $someValue)->will()->returnValue()->identicalTo($someValue)
    ->with(-1, $someValue)->will()->returnValue()->lessThan($someValue);
```

7.2 Exceptions

If you need to test an exception thrown by your method, you may do so with `throwException()` like so:

```
verify($calculator)->divide($someValue, 0)
    ->will()->throwException()->instanceOf(DivideByZeroException::class);
```

Just like with `returnValue()`, `throwException()` will call your method and then capture any exceptions it throws so that you can write assertion about the exception object. If your method does *not* throw an exception, `throwException()` will fail the test for you.

To inspect the exception further, you can drill into it by using the `withMessage()` and `withCode()` methods:

```
verify($calculator)->add(1, 'two')
    ->will()->throwException()->instanceOf(InvalidArgumentException::class)
    ->withMessage()->startWith('Invalid argument passed')
    ->withCode()->identicalTo(2);
```

7.3 Invokable Objects

If your subject is an object with an `__invoke()` magic method, you can write assertions about its return value or exceptions just like with other methods. Simply use `returnValue()` or `throwException()` after passing your subject to `verify()` and BeBat/Verify will invoke your object itself:

```
verify($subject)->will()->returnValue()->identicalTo('return value of __invoke()');
```

You can supply parameters for your subject using `with()` just like other methods:

```
verify($subject)
    ->with('invalid parameter')->will()
        ->throwException()->instanceOf(InvalidArgumentException::class)
    ->with('correct parameter')->will()
        ->returnValue()->identicalTo('correct parameter value');
```

EIGHT

EXTENDING

BeBat/Verify includes almost all the assertions built into PHPUnit, and all the ones from bebat/filesystem-assertions, but there may be additional assertions you need in your project. Depending on the number and complexity of assertions you want to add, BeBat/Verify includes two ways for you to extend it and add your own assertions.

8.1 Custom Constraint

Constraints are the building blocks for both PHPUnit and BeBat/Verify's assertions. It is possible to write your own constraints by extending PHPUnit's Constraint class.

To assert a constraint, pass it to BeBat/Verify's `constraint()` method after a conjunction, just like any other assertion. For example, if you had the package coduo/php-matcher installed:

```
use Coduo\PHPMatcher\PHPUnit\PHPMatcherConstraint;

use function BeBat\Verify\verify;

verify('{"name": "Norbert"}')->has()
    ->constraint(new PHPMatcherConstraint('{"name": "@string@"}'));
```

8.2 Custom Verifier

Using a custom constraint works well if your assertion is a one off and relatively simple. For anything more complicated though you should create your own *verifier* class. A verifier extends `BeBat\Verify\API\Base` and includes one or more assertion methods.

To use your verifier in an assertion chain, pass its class name to `withVerifier()`. BeBat/Verify will instantiate your verifier and pass it the subject and its name. If the constructor requires any additional arguments they can be passed to `withVerifier()`.

The `withVerifier()` method can also be used to switch between the value and file verifiers. For example, suppose you were testing a method that created a file and returned its path. If you wanted to write assertions about both the file contents and its name, you could do so by switching between verifiers with the `withVerifier()` method:

```
use BeBat\Verify\API\File;

// ...
```

(continues on next page)

```
verify($subject->writeFile())->will()->endWith('.log')        // assertion about the file␣
→path
    ->withVerifier(File::class)->contain('My Log Message'); // assertion about the file␣
→contents
```

For more details about writing your own verifier, see its *API documentation*.

VERIFIER API

You can add functionality to BeBat/Verify by creating a custom assertion class, or "verifier". Your verifier can then be swapped in using the `withVerifier()` method. All verifiers must extend `BeBat\Verify\API\Base`, which provides common functionality for assertion methods. This page describes the public and protected methods built into `BeBat\Verify\API\Base` that are most relevant to creating a verifier, although it is not a complete list of every method that class includes.

class `BeBat\Verify\API\`**Base**

> **assert**()
>
>> **Returns** `BeBat\Verify\API\Assert` (extends `PHPUnit\Framework\Assert`)
>
> Get an instance of PHPUnit's [Assert class](#). This class exposes much of PHPUnit's functionality for writing tests & assertions, such as causing a test to fail if an error occurs.
>
> **constraintFactory**()
>
>> **Returns** `BeBat\Verify\Constraint\Factory`
>
> The constraint factory is used to create constraints in BeBat/Verify. It includes most of the constraints from PHPUnit as well as those from [bebat/filesystem-assertions](#).
>
> **setAssert**(*$assert*)
>
>> **Parameters**
>>
>>> • **$assert** (`PHPUnit\Framework\Assert`) – An instance of PHPUnit's assertion object
>>
>> **Returns** `void`
>
> Inject an instance of `PHPUnit\Framework\Assert`. Useful for unit testing your verifier.
>
> **setConstraintFactory**(*$factory*)
>
>> **Parameters**
>>
>>> • **$factory** (`BeBat\Verify\Constraint\Factory`) – An instance of the BeBat/Verify constraint factory
>>
>> **Returns** `void`
>
> Inject an instance of `BeBat\Verify\Constraint\Factory`. Useful for unit testing your verifier.
>
> **constraint**(*$constraint*)
>
>> **Parameters**
>>
>>> • **$constraint** (`PHPUnit\Framework\Constraint\Constraint`) – Constraint to be applied

> **Returns** static

Apply a constraint to your verifier's subject. This is the simplest way to perform an assertion in your verifier.

performAssertion(*$constraint*, *$value*)

> **Parameters**
>
> - **$constraint** (PHPUnit\Framework\Constraint\Constraint) – Constraint to be applied
> - **$value** (mixed) – Value the constraint should apply to
>
> **Returns** static

Apply a constraint to a passed value. This method provides a bit more flexibility over *BeBat\Verify\API\Base::constraint* if there is some resolution required to determine the *actual* value a constraint should apply to.

performEqualToAssertion(*$actual*, *$expected*)

> **Parameters**
>
> - **$actual** (mixed) – The actual value under test
> - **$expected** (mixed) – Value $actual is expected to equal to
>
> **Returns** static

Apply an EqualTo() constraint on $actual with $expected. This method will take into account the various *modifiers* that apply to EqualTo(), including both *withoutCase()* and *withoutLineEndings()* simultaneously.

assertConstraint(*constraint*, *$value*)

> **Parameters**
>
> - **$constraint** (PHPUnit\Framework\Constraint\Constraint) – Constraint to be applied
> - **$value** (mixed) – Value the constraint should apply to
>
> **Returns** void

Perform a simple assertion with $constraint and $value. This method is useful for *interim* assertions about some value before your primary constraint (for example, asserting that a file exists before reading it and doing assertions about its contents). The assertConstraint() method does not take into consideration any modifiers or whether the current condition is positive or negative, it just applies $constraint to $value.

getActualValue()

> **Returns** mixed

Resolve the actual value of the subject. You may override this method in your verifier if there is some additional logic to resolving your subject's value, such as reading the result from an object property or function call.

resetParams()

> **Returns** void

Reset the modifiers to their default state and clear the description. This method will be called after performing an assertion. If your verifier includes custom modifiers you should override this method to set their value back to default, and call parent::resentParams().

9.1 Fluent Design

Your verifier should use a fluent interface, meaning all publicly available methods should return `self`. For your assertion methods, the easiest way to do this is to return a call to *BeBat\Verify\API\Base::constraint* with your assertion's constraint. If you need more flexibility with resolving your subject's value (such as reading it from a file) you may return *BeBat\Verify\API\Base::performAssertion* instead. Lastly, if your assertion is that two values are equal, you can us the *BeBat\Verify\API\Base::performEqualToAssertion* to simplify handling the various modifiers and edge cases that constraint supports. All three of these methods will also handle negative assertions for you, as well as resetting the classes internal state for the next assertion.

b

 h

INDEX